IDA PAPER P-2925

AN EXAMINATION OF SELECTED SOFTWARE TESTING TOOLS: 1993 SUPPLEMENT

Christine Youngblut

Bill Brykczynski, *Task Leader*

October 1993

Prepared for
Ballistic Missile Division Organization

Approved for public release, unlimited distribution: 12 May 1994.

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

IDA PAPER P-2925

AN EXAMINATION OF SELECTED SOFTWARE TESTING TOOLS: 1993 SUPPLEMENT

Christine Youngblut

Bill Brykczynski, *Task Leader*

October 1993

IDA

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.21

PREFACE

This paper was prepared for the Ballistic Missile Defense Organization (BMDO) as a follow-on effort for the Task Order "Software Testing of Strategic Defense Systems." It fulfills an objective of this subtask, to assist the BMDO in planning, executing, and monitoring software testing and evaluation research, development, and practice.

This paper is a supplement to IDA Paper P-2769, *An Examination of Selected Software Tools: 1992*. It reports the results of examining another 8 static and dynamic analysis tools in addition to the 27 testing tools originally presented in P-2769.

This paper was reviewed by Dr. Dennis Fife, a research staff member of the Institute for Defense Analyses.

FOREWORD

This report is a continuation of IDA Paper P-2769, *An Examination of Selected Software Testing Tools: 1992* [Youngblut 1992]. From Part I of P-2769, the introductory sections (Sections 1 through 3) also apply to the current report and, in the interests of space, have not been repeated. Section 4, "Approach and Methods," of P-2769 identified the examined tools and the examination approach. In this update, Table 1 identifies the tools recently examined and, for convenience, the original tool list is repeated in Table 2. An additional trial Ada program was included to demonstrate tool capabilities for testing concurrent Ada software, the Real-Time Temperature Sensor as described by Nielsen [1988].

Table 1.Tools Examined in this IDA Study

TOOL NAME	TOOL SUPPLIER	LANGUAGES SUPPORTED					TEST CAPABILITIES			
		Ada	C	C++	Fortran	Other	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis
ACT[a]	McCabe & Associates	√	√	√	√	√			√	√
Ada-ASSURED	GrammaTech, Inc.	√							√	
AdaTEST	Information Processing Ltd.	√							√	√
Battlemap[b]	McCabe & Associates	√	√	√	√	√			√	√
CodeBreaker	McCabe & Associates	√	√	√	√	√			√	
METRIC[c]	Software Research, Inc.	√	√	√	√				√	
McCabe Instrumentation Tool	McCabe & Associates	√	√	√	√	√			√	√
SLICE	McCabe & Associates	√	√			√				√

a. Used with McCabe Instrumentation Tool for graphical reporting of unit coverage.
b. Used with McCabe Instrumentation Tool for graphical reporting of design subtree coverage.
c. New component of the Software TestWorks (STW) toolset.

Sections in P-2769 on static and dynamic analysis are repeated here, extended with information for the newly examined tools. Since these new tools did not provide explicit support for test management or problem reporting, the corresponding sections from P-2769 have not been repeated. The findings remain unchanged and also are not repeated. Part II presented tool and supplier profiles for the previously examined tools. This has been updated to include information on the new tools. The remainder of Part II consists of tool examination reports that supplement the examination reports provided in P-2769.

Table 2.Tools Examined In the Previous IDA Study

TOOL NAME	TOOL SUPPLIER	LANGUAGES SUPPORTED					TEST CAPABILITIES			
		Ada	C	C++	Fortran	Other	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis
ADADL Processor	Software Systems Design	√	√		√				√	
AdaQuest	General Research Corp.	√							√	√
AutoFlow-Ada	AutoCASE Technology	√	√			√			√	
DDTs	QualTrak Corp.	+	+	+	+	+		√		
EDSA	Array Systems Computing, Inc.	√							√	
GrafBrowse	Software Systems Design	√	√		√				√	
LDRA Testbed	Program Analysers, Ltd.	√	√	√	√	√	F		√	√
Logiscope	Verilog, Inc.	√	√	√	√	√			√	√
MALPAS	TA Consultancy Services, Ltd.	√	√		√	√			√	
Metrics Manager	Computer Power Group, Inc.	+	+	+	+	+	√			
QES/Manager	Quality Engineering Software, Inc.	+	+	+	+	+	√			
QualGen	Software Systems Design	√	√		√				√	
S-TCAT	Software Research, Inc.	√	√		√	√			√	√
SQA:Manager	Software Quality Automation	+	+	+	+	+	√	√		
SRE Toolkit	Software Quality Engineering	+	+	+	+	+	√			
SoftTest	Bender & Associates	+	+	+	+	+				√
T	Programming Environments, Inc.	+	+	+	+	+				√
T-PLAN	Software Quality Assurance, Ltd.	+	+	+	+	+	√	√		
TBGEN	Testwell Oy	√	√	√						√
TCAT	Software Research, Inc.	√	√	√	√	√			√	√
TCAT-PATH	Software Research, Inc.	√	√	√	√	√			√	√
TCMON	Testwell Oy	√	√	√						√
TDGen	Software Research, Inc.	+	+	+	+	+				√
TSCOPE	Software Research, Inc.	+	+	+	+	+				√
TST	STARS Foundation Repository	√							√	√
Test/Cycle	Computer Power Group, Inc.	+	+	+	+	+	√	√		
TestGen	Software Systems Design	√	√						√	√

+ - *Language independent*
F - Future capability

Table of Contents

List of Figures

List of Tables

PART I
STUDY OVERVIEW

7. STATIC ANALYSIS

Static analysis is used to determine the presence or absence of particular, limited classes of errors, to produce certain kinds of software documentation, and to assess various characteristics of software quality. Unlike dynamic analysis, static analysis can sometimes be performed on incomplete or partly development products and does not necessitate costly test environments. It cannot, however, replace dynamic analysis, although it can be used to guide and focus dynamic testing. Previously, Table 1-1 and Table 1-2 identified 19 tools as supporting static analysis. The functions provided by these tools are summarized in Table 7-1. (For convenience, this and all subsequent tables use shading to highlight the newly examined tools.)

7.1 Complexity Analysis

Complexity measures can be put to various test-related uses. McCabe [1982] has developed a method, Structured Testing, that uses cyclomatic complexity to guide the selection of a minimum set of required paths to test. Complexity measures are also used to estimate the number of defects present in a piece of software, to identify pieces of code that are potentially defective, and to estimate needed development effort.

Models for estimating program complexity have been based on various characteristics of software structure and semantics. The best-known set of complexity measures are all applied at the program unit level. They are McCabe's cyclomatic complexity metrics and Halstead's software science metrics [1977]. Whereas cyclomatic complexity is control oriented, the Halstead metrics are text oriented. As well as variations on each of these measures, there are many other unit-level measures. In contrast, relatively few measures for assessing design-level complexity have been proposed. Perhaps the most common design-level measures are those developed by Mohanty [1976] that are based on a call graph and basic subtrees, a variation on cyclomatic complexity. Measures for assessing requirements complexity are similarly scarce and not supported by any of the examined tools. Table 7-2 identifies the different types of complexity measures that are provided.

Table 7-1. Static Analysis Capabilities of Examined Tools

TOOL NAME	Complexity Analysis	Data Flow Analysis	Control Flow Analysis					Information Flow Analysis	Standards Conf. Analysis	Quality Analysis	Cross-Reference Analysis	Browsing	Symbolic Evaluation	Specification Compliance	Pretty Printing
			Structure Analysis	Path Analysis	Code Statistics	Flowgraph Generation	Call Graph Generation								
ADADL Processor	√										√				√
AdaQuest				√					F		F				
AutoFlow-Ada						√	F								√
EDSA		√		√							√	√			√
GrafBrowse							√					√			
LDRA Testbed	√	√	√	√	√	√	√	√	√		√				√
Logiscope	√		√	√	√	√	√			√		√			
MALPAS	√	√	√	√		√		√					√	√	
QualGen										√					
S-TCAT			√		√		√					√			
TCAT			√		√	√						√			
TCAT-PATH	√		√	√	√	√						√			
TST				√					√						√
TestGen	√			√		√									
ACT	√			√	√	√									
Ada-ASSURED									√			√			√
AdaTEST	√	F			√	F					F				
Battlemap	√			√	√	√	√				√	√			√
CodeBreaker	√		√												
METRIC	√				√						√				
McCabe Instrumentation Tool	√														

F - Future capability

Twenty years of theoretical and empirical evaluations have failed to produce consistent, hard evidence of the accuracy of particular measures or on the respective value of other measures. Consequently, these measures should be used as indicators rather than as absolute measures of software properties.

Table 7-2. Supported Complexity Measures

TOOL NAME	Unit Level													Integ Level	
	Basic Blocks (#)	Basic Block (Avg. Len.)	Cyclomatic Complexity	Essential Complexity	Intervals (Ord. 1)	Intervals (Max. Ord.)	Control Flow Knots	Essential Knots	Paths (#)	Paths (Avg. Length)	Paths (Max/Min)	Iteration Groups (Max.)	Halstead's Metrics	Basic Subtrees (S1)	Mohanty's Metrics
ADADL Processor			√												
LDRA Testbed	√	√	√	√	√	√	√	√	√						
Logiscope			√	√									√		√
MALPAS			√												
TCAT-PATH	√		√	√					√	√	√	√			
TestGen			√											√	
ACT			√	√									√		
AdaTEST			√	F									√		
Battlemap			√	√									√	√	
CodeBreaker			√	√											
METRIC			√	√									√		
McCabe Instrumentation Tool			√											√	

F - Future capability

7.2 Data Flow Analysis

Data flow analysis is based on consideration of the sequences of events that occur along the various paths through a program. It is used to detect data flow anomalies, of which three types are commonly recognized: (1) a variable whose value is undefined is referenced, (2) a defined variable is redefined before it is referenced, or (3) a defined variable is undefined before it is referenced. While the first of these indicates an actual program defect, the second and third types of anomaly may indicate questionable variable use rather than specific defects.

Since the analysis technique assumes that all paths through the program are feasible, some reported anomalies may be superfluous. Data flow analysis also can be used to categorize procedure parameters as referenced only, defined only, both defined and referenced, or not used.

LDRA Testbed, MALPAS, and EDSA support static data flow analysis. LDRA Testbed performs weak data flow analysis to identify data flow anomalies of the types mentioned above. It also analyzes procedures calls across procedure boundaries to report on procedure parameter use. MALPAS refines the classification of data flow anomalies. For example, a data variable that is redefined before it is referenced may be classified as either an instance where data is written twice without an intervening read, or as data being written with no subsequent access on a given path. Given a list of procedure input and output parameters, MALPAS compares these with the classes of data to produce a table of possible errors. EDSA uses interactive data flow analysis to facilitate program browsing.

7.3 Control Flow Analysis

Control flow analysis is a process of examining a program structure and identifying major features such as entry and exit points, loops, unreachable code, and paths through a program. This information can be used to determine program complexity and to aid in planning a dynamic test strategy. It can help to decide on strategies for further analysis, for example, to identify where it might be beneficial to partition the code to reduce the number of paths and, hence, facilitate semantic analysis. The results of control flow analysis can also be used to prepare a diagram of the program structure that can aid a user in documenting and understanding a piece of software.

Control flow analysis is provided by the majority of tools that support static analysis. MALPAS, TestGen, LDRA Testbed, and the TCAT family all report on unreachable paths. These may be generated as a result of program syntax, for example, as a result of *end if* statements, or the position of a *return* statement. Even though they do not necessarily imply a defect, the occurrence of unreachable paths should be checked. Some of the examined tools go farther. LDRA Testbed, for example, also reports on unreachable branches and other structural units.

Several of the tools use control flow analysis to generate a graphical representation of a program's structure as a logical flow chart or directed graph. This allows visual inspection of program structure and complexity, and can facilitate program understanding at the unit level. ACT, AutoFlow-Ada, Battlemap, LDRA Testbed, Logiscope, TCAT, and TCAT-PATH all generate fairly sophisticated graphs of a program's structure. AutoFlow-Ada, in particular, provides a user with considerable flexibility in generating a high-quality graphical flow chart. TestGen uses textual facilities to produce a more primitive graph. Although MALPAS does not directly produce a directed graph, its list of nodes, with identification of successor and predecessor nodes, helps a user to draw this graph. Graphical representation of the calling relationship between program units also facilitates program understanding.

Battlemap, GrafBrowse, LDRA Testbed, Logiscope, and S-TCAT generate call graphs or call trees.

The identification of paths through a program is useful for estimating the resources needed for dynamic analysis and then guiding this testing. ACT, AdaQuest, LDRA Testbed, Logiscope, MALPAS, TCAT-PATH, TST, and TestGen all provide this capability. Even more useful, ACT, LDRA Testbed, Logiscope, and TestGen explicitly identify the values of logical conditions necessary to cause particular paths to be followed (in the case of ACT, these are the set of paths required for McCabe's Structured Testing method). Battlemap identifies the design subtrees that form the integration structure of a program and also the values of logical conditions necessary to cause particular subtrees to be followed (in this case, the tool distinguishes between the full set of design subtrees and the more limited set of cyclomatic subtrees).

ACT, AdaTEST, Battlemap, Logiscope, METRIC, TCAT, TCAT-PATH, and S-TCAT report on various code statistics. These statistics range from measures such as the total number of dynamic memory allocations, to measures of the average, minimum, and maximum path length. EDSA provides interactive control flow analysis to facilitate browsing along program paths.

MALPAS, LDRA Testbed, and Logiscope perform structure analysis to verify a program's conformance to the principles of structured programming. Here LDRA Testbed matches templates of acceptable structures with the directed graph of a program on a module by module basis. Matching structures are successively collapsed to a single node until either a single node is left, indicating a structured program, or an irreducible state, indicating an unstructured program. MALPAS and Logiscope perform a similar reduction to evaluate the structure.

CodeBreaker is unusual in that it is designed to serve a single purpose. It uses structure analysis to compare the design structures of two (sub)programs or to compare the control structures of two individual modules. Any discrepancies between the two are identified.

7.4 Information Flow Analysis

Information flow analysis is used to examine program variable interdependencies. This helps to isolate inadvertent or unwanted dependencies, to indicate how a program can be broken down into subprograms, and to identify the scope of program changes. For security applications, it can be used to aid the identification of spurious or unknown code. Additionally, it supports dynamic testing by identifying which inputs need to be exercised to affect which outputs.

Both LDRA Testbed and MALPAS provide this capability. Currently LDRA Testbed is limited to identifying backward dependencies on a procedure by procedure basis and characterizes variables as strongly or weakly dependent. Future versions of LDRA Testbed will include forward dependencies to identify variables that can be affected by a particular input variable. It will also support information flow dependence assertions to allow comparison of expected dependencies with actual dependencies.

MALPAS identifies all of a program's inputs and examines each executable path to identify dependencies for each output variable. These dependencies include the input variables, constants, and conditional statements on which it depends. It reports on program unit inputs and outputs, which may be more than those passed as parameters. MALPAS also identifies redundant statements.

7.5 Standards Conformance Analysis

Auditors are used to check the conformance of a program to a set of standards. For Ballistic Missile Defense (BMD) software, the Software Productivity Consortium (SPC) *Ada Quality and Style: Guidelines for Professional Programmers* [SPC 1991] defines the required standards. Ada-ASSURED supports these guidelines, as does ADAMAT, discussed in the Ballistic Missile Defense Organization (BMDO) Computer Resources Working Group (CRWG) study.

LDRA Testbed checks conformance to a set of standards of interest to the programming community; this includes much of the Safe Ada Subset. Individual standards can be disabled and the user can weight particular standards or specify acceptance limits, where appropriate. TST reports on conformance to a set of portability standards.

7.6 Quality Analysis

As already mentioned, several tools report on particular quality characteristics such as complexity and compliance with standards. There are, however, many other quality characteristics that provide insight into code maintainability and portability, for example.

One of the examined tools, Logiscope, employs the Rome Air Development Center (RADC) quality metrics model to allow user-defined quality measurement at three levels of abstraction [RADC 1983]. At the lowest level of the model, the user can defined upper and lower bounds for a predefined set of primitive metrics. Logiscope distinguishes between unit-level metrics and architectural metrics, reporting on both. The user can then specify algorithms to weight and combine the primitive metrics into composite metrics. These composite metrics are, in turn, used to define quality criteria that allow classifying components as, for example, accepted or rejected, based on their computed quality values.

QualGen analyses both design and code complexity and currently interfaces with Lotus 1-2-3 for quality reporting. It provides some 200 primitive metrics which, via Lotus, can be combined into user-defined higher level measures. AdaTEST provides a similar facility through the use of.csv files. Software Systems Design, the developer of QualGen, is currently mapping the correspondence of QualGen metrics to the SPC Ada style guide.

7.7 Cross-Reference Analysis

The information acquired from cross-referencing program entities serves many purposes. Perhaps one of the most important of these is identifying the scope of a program change or aiding in the diagnosis of a software failure.

The ADADL Processor provides extensive cross-referencing capabilities. It reports on the cross-referencing between program units, objects, and types. It also reports on the occurrence of *with* and *pragma* statements; the occurrence of interrupts, exceptions, and generic instantiations; and the use of program unit renaming. LDRA Testbed cross-references all data items and classifies them as global, local, or parameter and also cross-references procedure usage. METRIC identifies all generic units, the number of times they are instantiated, and the instantiating unit. Through its browsing capabilities, EDSA provides interactive cross-referencing of data items and Ada objects. Battlemap cross-references program units in terms of their calls-to and called-by relationships, it also provides context and index listings to identify the files in which particular units are stored.

7.8 Browsing

A browser facilitates program understanding by allowing the user to create and present different views of the software. This may include views that show the same piece of software at different stages of development and views that omit some information in order to focus on other details. A browser also may provide the user with the ability to follow the control flow or data flow. These capabilities may be used for several purposes, for example, to aid in reviewing a program or in diagnosing the cause of a software failure.

EDSA focuses on browsing source code at the unit level; it allows browsing forward or backward via data flow or control flow. The user can construct views that suppress or omit irrelevant code details to help the user to focus on the concern at hand. Special annotations are available to keep track of the progress of formal code verification. Ada-ASSURED provides similar capabilities with navigation based on control structure. Graf-Browse chiefly operates at the integration level. Here the user can move through graphical invocation hierarchies (or declaration or call-by hierarchies), pulling up the relevant pieces of code as required. Battlemap and the TCAT family of coverage analyzers also allow moving between graphical depictions of program and module structure and the associated source code.

Although not examined in the course of this work, the new version of Logiscope also supports source code browsing.

7.9 Symbolic Evaluation

This type of static semantic analysis provides a more complete examination of a program's operation. Instead of actual input data, symbols such as variable names are used to simulate program execution. This allows the reporting of the mathematical relationships between inputs and outputs for each semantically possible path. It has three primary uses. The relationships can be compared against a program specification to check for consistency. The identified path condition, together with the expression detailing the set or range of input data which causes this path to be executed, supports test data generation. Finally, the relationships can aid in determination of the expected output for a set of test data. Only MALPAS provides this very useful capability.

7.10 Specification Compliance Analysis

Specification compliance analysis takes semantic analysis a step further by automatically comparing a program against its formal specification to identify deviations. This type of analysis is very powerful, but requires additional work on the behalf of the user.

Here again, MALPAS was the only examined tool that provides this capability. It requires program specification details to be embedded in its intermediate language. (These details may already be available if a formal specification language such as Z or the Vienna Development Method (VDM) is used in the development effort.) The output of the compliance analyzer is a set of threat statements that, if the program does not meet the specification, presents the relationships between inputs that cause a deviation to occur.

7.11 Pretty Printing

A useful documentation capability, pretty printing is provided by the ADADL Processor, Ada-ASSURED, AutoFlow-Ada, Battlemap, EDSA, LDRA Testbed, and TST.

8. DYNAMIC ANALYSIS

This section reports on the capabilities provided by the examined tools for dynamic analysis where software is evaluated based on its behavior during execution. Dynamic analysis is the primary method for validating and verifying software. Additionally, it is the source of much of the information used in monitoring testing progress and software quality. Traditionally an unstructured and labor-intensive activity, dynamic analysis is a significant cost driver. Table 8-1 identifies the particular functionality provided by the examined tools.

8.1 Assertion Analysis

An assertion is a logical expression specifying a program state that must exist, or a set of conditions that program variables must satisfy, at a particular point during program execution. Assertion analysis is used to determine whether program execution is proceeding as intended. In some cases, it may be desirable to leave assertions permanently in the code to provide a self-checking capability. When present in code, even if commented out, assertions can provide valuable documentation of intent.

Of the examined tools, only LDRA Testbed currently supports dynamic assertion analysis. Assertions are embedded in Ada comments and can be used to (1) specify pre- and post-conditions for a section of code, (2) check whether inputs satisfy pre-determined ranges, and (3) check whether loop and array indices are within bounds. Should any assertion fail, a user-tailorable failure handling routine is executed. Assertion checking can be switched on or off, allowing assertions to remain permanently in the code.

8.2 Coverage Analysis

Coverage analysis is the process of determining whether particular parts of a program have been exercised. Its importance is illustrated by academic studies and the experience of the software testing industry that have shown that the average testing group that does *not* use a coverage analyzer exercises only 50% of the logical program structure. As much as half the code is untested and therefore many errors may go undetected at the time of release.

Table 8-1. Dynamic Analysis Capabilities of Examined Tools

TOOL NAME		Coverage Analysis				Prof-iling					Test Data Generation						
	Assertion Analysis	Structural (Unit)	Structural (Int.)	Data Flow	Functional	Execution Counts	Instruction Tracing	Timing Analysis	Task Analysis	Test Bed Generation	Structural	Functional	Parameter	Grammar-based	Test Data Set Analysis	Dynamic Flow Graph	Dynamic Call Graph
AdaQuest	F	√				√		√	F								
LDRA Testbed	√	√		F		√	√				√				√	√	√
Logiscope		√	√			√					√					√	√
SoftTest					√							√					
S-TCAT			√			√											
T					√							√					
TBGEN										√							
TCAT		√				√											
TCAT-PATH		√				√					√						
TCMON		√				√		√									
TST						√	√			√			√				
TDGen														√			
TSCOPE[a]		√	√													√	√
TestGen		√				√					√						
ACT[b]		√														√	
AdaTEST	F	√	√	F		√	√	√		√	√					F	
Battlemap[c]		√	√													√	√
McCabe Instrumentation Tool		√	√													√	√
SLICE							√									√	√

a. Used with TCAT, TCAT-PATH, or S-TCAT to animate coverage results.
b. Used with McCabe Instrumentation Tool for graphical reporting of unit coverage.
c. Used with McCabe Instrumentation Tool for graphical reporting of design subtree coverage.

F - Future capability

By identifying those parts of a program that have not yet been executed, a coverage analyzer can help to ensure that all code is exercised, thus increasing confidence in correct software operation. By measuring the coverage achieved during execution with particular set(s) of test data, these tools also provide a quantitative measure of test completeness. Some tools also aid in determining the test data needed to increase the coverage. Although coverage analyzers do not directly measure software correctness, they are valuable tools for guiding the testing process and monitoring its progress.

There are two basic types of coverage analyzers. Intrusive analyzers instrument code with special statements, called probes, that record the execution of a particular structural program element. The addition of extra code in the program incurs both a size and timing overhead. The alternative, non-intrusive analyzers, requires special hardware and is not addressed in this report.

8.2.1 Structural Coverage Analysis

Several levels of structural test coverage have been proposed. The basic levels for unit testing are statement, branch, and path coverage which require, respectively, each statement, branch, or path to be executed at least once. These coverage levels impose increasingly stringent levels of testing with statement coverage being the weakest and path coverage the strongest. Since path coverage can be difficult to achieve, various additional levels that lie between branch and path coverage have been proposed. The best known of these additional levels are McCabe's Structured Testing and Linear Code Sequence and Jumps (LCSAJs) [Hennell 1976].

Although unit-level measures can be applied during integration and system testing, they do not provide the additional information that is pertinent at these levels. During integration testing, for example, a measure of the extent to which the relationships between calling and called units has been executed is useful. Functional measures provide a more appropriate measure of test coverage for system testing (see Section 8.2.3).

Table 8-2 summarizes the structural coverage analysis features of the examined tools. As shown in this table, the examined tools vary considerably in the support they provide. The requirements for a test driver to execute the instrumented program an example of one of these differences. LDRA Testbed and TCMON automatically generate this test driver, as does TestGen under certain circumstances. The generated test drivers also differ. For example, TCMON provides a command-driven test driver that allows the user to explicitly control the handling of generated trace files. Where necessary, both LDRA Testbed and TCMON allow special actions so that this interface can be omitted. There are other significant differences. For example, LDRA Testbed provides different handling of trace data to support host and target testing. It also separates out the data collected from a concurrent program to allow separate reporting for each task. AdaTEST provides an example of another significant difference between these tools; it performs coverage analysis at test execution time, obviating the need for post-processing of large trace files.

8.2.2 Data Flow Coverage Analysis

Data flow coverage has been proposed as another measure of test data adequacy. While the traditional structural coverage testing approach is based on the concept that all of the code must be executed to have confidence in its correct operation, data flow testing is based on the concept that all of the program variables must be exercised.

While there are several tools that provide this capability for C programs, production quality tools for data flow testing of Ada code are not yet available. The data flow testing capability of LDRA Testbed, however, is currently under beta testing and AdaTEST is expected to provide this capability in the near future.

Table 8-2. Structural Coverage Analysis Characteristics

TOOL NAME	Unit-Level Coverage								Reporting									
	Statement Coverage	Branch Coverage	LCSAJ Coverage	Condition Coverage	McCabe Coverage	Path Coverage	User-specified Coverage	Integration Coverage	Accumulate Coverage	Animation of Coverage	Frequency Distribution	Warning Level	Identify (Not) Hit Items	Inst. Files Differently	Requires Driver	Limit Instrumentation	Host/Target Support	User-control of Trace Files
AdaQuest		√					√		√		√		√			√		√
LDRA Testbed	√	√	√	√					√	√	√	√	√	√	√		√	
Logiscope	√	√	√					√	√		√		√	√			√	
S-TCAT								√	√		√		√	√		√		√
TCAT		√		√					√		√		√	√		√		√
ТCAT-PATH						√			√				√	√		√		
TCMON	√			√					√		√	√	√	√	√			√
TSCOPE[a]		√				√		√		√	√							
TestGen	√	√			√	√			√				√		√			√
ACT[b]		√			√				√				√					
AdaTEST	√	√		√				√	√	F		F	√	√	√	√	√	√
Battlemap[c]								√	√				√					
McCabe Instrumentation Tool	√	√			√	√		√	√			√	√			√		

a. Used with TCAT, TCAT-PATH, or S-TCAT to animate coverage results.
b. Used with McCabe Instrumentation Tool for graphical reporting of unit coverage.
c. Used with McCabe Instrumentation Tool for graphical reporting of design subtree coverage.

F - Future capability

8.2.3 Functional Coverage Analysis

Functional coverage, sometimes called requirements coverage, provides a measure of the extent to which tests have caused execution of the functions that the software is required to perform. Unlike structural tests, functional tests can determine problems such as the absence of needed code.

Two of the examined tools assess the functional coverage of tests. SoftTest provides a measure of test adequacy in terms of the number of tested functional variations with respect to the number of those testable. T provides a measure of test adequacy based on requirements coverage using user specified pass/fail results. An additional test comprehensiveness measure considers requirements coverage, input domain coverage, output range coverage and, optionally, structural coverage, where each factor can be user weighted.

8.3 Profiling

Profiling provides a trace of the flow of control during software execution. This information can aid in locating the cause of a failure and the position of the associated defect. Of the examined tools, AdaTEST, LDRA Testbed, and TST provide this capability as an optional feature. AdaTEST allows the user to specify the level of tracing required. The user can request tracing of unit calls, tracing of decision calls, or full tracing at the statement level. LDRA Testbed and TST provide statement level tracing. In the case of LDRA Testbed, however, the Testbed may override the user request if the resulting display exceeds a preset limit. SLICE operates in conjunction with the McCabe Instrumentation Tool and Battlemap to identify, both textually and graphically, the particular program elements exercised during a program execution in one or more program units. AdaTEST allows the user to specify the level of tracing required. The user can request tracing of unit calls, tracing of decision calls, or full tracing at the statement level.

In general, the majority of computing time is incurred by only a few program segments. This may be because these segments are called frequently, are computationally intensive, or both. When a program needs to be optimized, therefore, it is more efficient to start by identifying where the majority of computing time is spent so that the optimization effort can be appropriately focused. Information on the number of times particular program segments are executed can aid this determination. The coverage analysis tools all give the number of times examined program elements are executed; some additionally identify the number of times each program unit is invoked.

8.4 Timing Analysis

Timing analysis serves several purposes. These range from supporting the validation of requirements that impose specific timing constraints on software functions to identifying particular program units that consume a significant proportion of computing time.

AdaQuest, AdaTEST, and TCMON provide timing analysis. Both AdaQuest and TCMON offer the flexibility of user-specified placement of timers, and measurement using either clock or wall time. TCMON additionally allows a user to request automatic timer instrumentation at the program unit level. This tool reports on the placement of timers (and any counters used for structural coverage analysis) to provide information that can be used to estimate the influence of instrumentation statements on measured time. AdaTEST provides the user the capability to start, stop, and reset a timer that captures execution time to the resolution of the Ada CALENDAR.CLOCK.

8.5 Test Bed Generation

Unit and integration testing require the ability to invoke the appropriate modules, passing necessary inputs and capturing the actual outputs so that they can be compared against expected outputs. Integration testing may proceed in either a top-down or bottom-up manner. Top-down testing starts with the most abstract, or high-level modules, and requires the use of *stubs* to represent those modules called by the module under test. In bottom-up testing, the most detailed, or lower-level, modules are tested first. Here test *drivers* are required to simulate the modules that invoke the modules under test. Development of such test drivers and stubs can be complex and greatly facilitated by automated support. In addition to eliminating the need for much manual labor, automatic generation also promotes a standardized testing environment.

LDRA Testbed, TCMON, and TestGen all generate the test drivers needed for execution of an instrumented program. These are, however, very limited drivers primarily intended to handle the trace files used to collect coverage details. Of the examined tools, AdaTEST, TBGEN, and TST are the only ones that provide test control via some form of command language, and only AdaTEST and TBGEN support stub generation. Table 8-3 summarizes the test bed generation characteristics of these three tools.

8.6 Test Data Generation Support

Dynamic analysis requires software to be executed with a set of test data. The resulting outputs are then captured and compared with the outputs expected for the given input data. The traditionally manual and labor-intensive method of preparing test data has typically limited the extent of testing. Although the available tools do not totally replace the human effort required, they can make a substantial reduction to the amount of human labor needed.

Table 8-3. Test Bed Generation Characteristics

TOOL NAME	Driver Generation	Stub Generation	Interactive Operation	Script-based Operation	Control Included Entities	Support Table-driven Test	Command Language								Record Keeping			
							Variable Checks	Call Checks	External Event Checks	Exception Checks	Timer Checks	Test Bed Variables	Control Constructs	Breaks	Execution Log	I/O Tracing	Execution Statistics	User Input Recording
TBGEN	√	√	√	√	√		√			√		√	√	√	√	√	√	√
TST	√						√											√
AdaTEST	√	√		√	√	√	√	√	√	√	√	√	√		√		√	

As mentioned above, dynamic analysis requires comparing expected results against actual results to determine the success or failure of a test. Determining expected results is another traditionally manual and difficult task. Research into tools, called *oracles*, to automate this task has been ongoing for many years. As yet, however, symbolic evaluators (see Section 7.9) come the closest to supporting this capability.

8.6.1 Structural Test Data Generation

During testing, there are occasions where it is necessary to determine the test data that will cause a specific branch or path to be executed. This occurs, for example, when it is necessary to attain a specified level of structural coverage and existing test data has not executed some structural elements.

Support for this activity is available at two levels. AdaQuest and TCAT explicitly identify the program segments that comprise particular program branches and paths. LDRA Testbed, Logiscope, TCAT-PATH, and TestGen provide the same information and, additionally, explicitly identify the conditions required to cause each structural element to be executed.

8.6.2 Functional Test Data Generation

Functional tests can be derived from a requirements specification using three categories of methods: (1) algorithmic techniques such as cause-effect graphing, equivalence class partitioning, and boundary value analysis; (2) heuristic techniques including fault directed testing and the traditional error guessing; and (3) random techniques that employ random generation of test data.

T supports all these techniques. Additionally, it is capable of incremental test data generation, that is, tests can be generated for software changes only. T is the only examined tool that produces test data values ready for immediate use in testing.

SoftTest supports cause-effect graphing to compile a database of input conditions for each unique function. The user then works from these conditions to determine the necessary test data. In those cases where identified functions are not directly testable, for example, because results produced by one function may be obscured by other functions, SoftTest identifies intermediate results that, if observable, would enable otherwise obscured functions to be tested.

8.6.3 Parameter Test Data Generation

Thorough test coverage at the integration level requires that each subprogram be executed over a range of parameter values. Of the examined tools, only TST provides automated generation of test data for certain types of subprogram parameters. This generation occurs in one of two forms. The user can specify that all possible values for a parameter be generated (or first and last values for floating point numbers). Alternatively, the user can request that these values are divided into a number of partitions and that the first, middle, and last values from each partition be selected.

8.6.4 Grammar-based Test Data Generation

In those cases when the test data is simply structured, and this structure is amenable to description, grammar-based test data generation allows rapid, automated generation of large amounts of test data. This capability is particularly useful in random testing.

TDGen provides this functionality. Test data is generated according to location-specific data, uniformly distributed data, or value-factored data. TDGen can generate data randomly, sequentially, or according to a user specification.

8.7 Test Data Analysis

Two types of test data analysis are considered here. In the first case, test data sets are analyzed to identify which test data sets execute which lines of code. When particular lines of code are changed, this information shows which test data sets are affected by the change and must be rerun. The second type of test data analysis detects and reports on redundant test data sets. This identifies test data sets that are essentially equivalent in effect and, therefore, can be eliminated to reduce testing cost without affecting test effectiveness.

LDRA Testbed is the only identified tool that supports these capabilities. The analyses are performed on data collected during structural coverage analysis.

8.8 Dynamic Graph Generation

A visual representation of the execution flow of a program can aid in understanding that program and diagnosing the cause of failures. ACT provides this capability at the unit level, whereas Battlemap, LDRA Testbed, and Logiscope provide it at both the unit and integration levels. TSCOPE uses the outputs of TCAT or TCAT-PATH to animate the execution coverage on a directed graph; and the output of S-TCAT can be used to animate coverage on a call tree representation of the program under test. SLICE uses the outputs of the McCabe Instrumentation Tool and Battlemap to highlight the path taken by a particular execution.

PART II
TOOL EXAMINATION REPORTS

10. INTRODUCTION

This part of the report describes the selected tools in terms of their usage. Tools are grouped by supplier and the report details the operating environment and the functionality provided. Where applicable, price information, accurate at the time of examination, is also included. Each description is supported with observations on ease of use, documentation and user support, and Ada restrictions. Problems encountered during the examinations provided insight into the reliability and robustness of each tool. Each description is accompanied by sample outputs.

Table 10-1 summarizes the details given for each tool. It also identifies available bridges between testing tools and CASE systems. Table 10-2 presents relevant supplier data.

Table 10-1 . Tool Profiles

TOOL NAME	TOOL STATISTICS				OPERATING ENVIRONMENT												LINKS TO OTHER TOOLS	
					Machine			O/S										
	First Marketed	Examined Version	Number Users/Sites	Starting Price	PC Machines	Workstations	Other	Unix	VMS	DOS	Windows Supported	Network Version	Graphics Capability	Import Formats	Export Formats	User Interface	Testing Tool Bridges	CASE/Other Bridges
ADADL Proc.	1984	5.3E	>200 S	$5,000	√	√	√	√	√	√	O					M		
AdaQuest	1991	1.1	<5 U	$6,500		√	√		√							C		CASE framework
AutoFlow	1992	1.02	<4,000 U	$9,950	√			F		√	F		F		ASCII, HPGL, PIC, Postscript	C		ADW, IEW
DDTs	1989	2.1.6	>100 S	$6,000		√	√	√			F	√		Converters	Converters	B	via QTET	HP Softbench, RCS, SCCS
EDSA	1991	2.0	<10 U	$3,750	√	√	√	√	√	√	√					M		RCS, SCCS
GrafBrowse	1991	2.2.2	<10 S	$5,500	√	√	√	√	√	√	√		√		Postscript, HPGL	M		
LDRA Testbed	1974	4.8.01	>400 S	$12,000	√	√	√	√	√	√	O	√	√	Defined formats	Defined formats, Postscript	M	TBGEN	Mascot, STP, System Engineer, Teamwork., & others
Logiscope	1985	3.2	>5,000 U	$14,000		√	√	√	√		√		√	ASCII	ASCII, DecWrite, HPGL, Interleaf, Postscript, pcl, imPRESS, Genicom	B		DecFuse, HP Softbench, Software Back Plane
MALPAS	1986	5.1	>50 U	$60,000			√		√							C		
Metrics Manager	1989	2.02	>30 S	$14,950	√					√	F		√	ASCII		M	Test/Cycle	Project Manager Workbench
QES/Manager	1991	2.2	>50 U	$2,500	√					√	F	√		Converters	Converters	M	QES/Architect	

Table 10-1 continued. Tool Profiles

TOOL NAME	TOOL STATISTICS				OPERATING ENVIRONMENT												LINKS TO OTHER TOOLS	
					Machine			O/S										
	First Marketed	Examined Version	Number Users/Sites	Starting Price	PC Machines	Workstations	Other	Unix	VMS	DOS	Windows Supported	Network Version	Graphics Capability	Import Formats	Export Formats	User Interface	Testing Tool Bridges	CASE/Other Bridges
QualGen	1988	1.1	<10 S	$4,000	√	√	√	√	√	√	O		√		Lotus	M		
S-TCAT	1987	1.3	>2,500 U	$4,900[b]	√	√		√		√	O	√	√			B	TSCOPE	HP Softbench, DEC FUSE
SQA:Manager	1990	2.0	>100 U	$995	√	√		√		√	√	√	√	Converters		M	SQA:Robot	
SRE Toolkit	1990	3.11	>200 U	$995[a]	√	√		√		√			√	Defined formats	PIC	C		
SoftTest	1987	3.1	>100 U	$2,500	√					√						M	AutoTester, Gate, Automator qa, SQA:Robot, TestPro, V-Test, X/TestRunner, Microsoft Test for Windows, Workstation Interactive Test Tool for OS/2	
T	1987	3.0	>300S	$7,000	√	√	√	√	√	√	√	√			ASCII, IEEE 1175	B	AutoTester, Ferret, XRunner	Excelerator, IEF, StP, Teamwork
T-PLAN	1989	2.0	>120U	£9,500	√	√	√	√		√	O	√		DIF, dBASE, Lotus	DIF, dBASE, Lotus	M	Automator qa	
TBGEN	1986	3.1	>30 S	$2,850	√	√	√	*	√	√	O					C	LDRA, TCMON	DDC-I CASE Toolbox[c]
TCAT	1986	1.3	>2,000 U	$4,900[b]	√	√		√		√	O	√	O			B	TSCOPE	HP Softbench, DEC FUSE
TCAT-PATH	1988	8	>2,500 U	$4,900[b]	√	√		√		√	O	√	O			B	TSCOPE	HP Softbench, DEC FUSE
TCMON	1986	2.2	>30 S	$2,300	√	√	√	*	√	√	O					C	TBGEN	DDC-I CASE Toolbox[c]
TDGen	1990	3.2	~500 U	$500	√	√		√		√	O		O			B		

Table 10-1 continued. Tool Profiles

TOOL NAME	TOOL STATISTICS				OPERATING ENVIRONMENT												LINKS TO OTHER TOOLS	
					Machine			O/S										
	First Marketed	Examined Version	Number Users/Sites	Starting Price	PC Machines	Workstations	Other	Unix	VMS	DOS	Windows Supported	Network Version	Graphics Capability	Import Formats	Export Formats	User Interface	Testing Tool Bridges	CASE/Other Bridges
TSCOPE	1989	1.2	>2,500 U	$4,900[b]		√		√			√	√	√			M	TCAT, S-TCAT, TCAT-PATH	
TST	1989	2.0	>3	Gov.	√		√		√	√						M		
Test/Cycle	1990	3.02	>10 S	$3,500	√					√	√		√		ASCII, Post-script, GKS, Tektronix	M	Metrics Manager	Ad/Cycle, Project Manager Workbench
TestGen	1984	2.2.2	>1,000 U	$4,600	√	√	√	√	√	√	O		O			M		Excelerator, StP, Teamwork
ACT	1987	V19920601-1	>1,000 U	$11,500	√	√	√	√	√	√	√		√			B	Casebridge	Softbench, WorkBench, Teamwork, StP
Ada-ASSURED	1992	1.0	>20 S	$1,495		√		√			√					G		Softbench
AdaTEST	1991	2.3/2.1	>15 S	£9,500	√	√	√	√	√	√		√			.csv files	C		
Battlemap	1988	V19920601-1	>1,000 U	$21,500		√	√	√	√		√		√		HPGL, Post-script, HP Laserjet	B	Casebridge	Softbench, WorkBench, Teamwork, StP,
Code Breaker	1992	V19920601-1	>500 U	$5,000	√	√	√	√	√	√						B		
METRIC	1993	2.10	>100 U	$4,000[a]		√		√			√	√	√		ASCII	B		
McCabe Instr. Tool	1991	V19920601-1	>1,000U	$5,000	√	√	√	√	√	√						B		
SLICE	1991	V19920601-1	>1,000U	b	√	√	√	√	√	√	√		√			B	Casebridge	Softbench, WorkBench, Teamwork, StP

a. Price quoted is for the standalone product, burdled with other STW tools price can be reduced to $775.

b. Sold in conjunction with McCabe Instrumentation Tool.

F- Capability under development
U - Users
S - Sites
O - Optional

M- Menu drive
C - Command driven
B - Menu and command driven

* - Version marketed by DDC International
a - Price for 3-day training course
b - Sold as group

Table 10-2 . Supplier Profiles

SUPPLIER NAME	SUPPLIER STATISTICS			SERVICES PROVIDED							Examined Tools	RELATED SUPPLIER TOOLS						
	Phone Number	Supplier Size	Established	Consultancy	Training	Test Planning	Test Performance	User Group	Newsletter	Hot-Line Support		Tool Name	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis	CASE
Array Systems Computing, Inc.	(416) 736-0900	<50	1981	√	√		√		√	√	EDSA							
AutoCASE Technology	(408) 446-2273	<10	1988	√						√	AutoFlow-Ada							
Bender & Associates	(415) 924-9196	<10	1977	√	√	√				√	SoftTest							
Computer Power Group, Inc.	(708) 574-3030	>3,000	1968	√	√	√	√	√	√	√	Test/Cycle, Metrics Manager							
General Research Corp.	(805) 964-7724	~1,000	1960s	√	√		√				AdaQuest	J7AVS			√	√		
												RXVP80			√	√		
GrammaTech, Inc.	(607) 273-7340	<10	1988	√	√				√	√	Ada-ASSURED	Synthesizer Gen.			√			√
IPL Information Processing Ltd.	+44 0225 444888	>200	1979	√	√	√	√			√	AdaTEST	CANTATA			√	√		
McCabe & Associates	(410) 995-1075	<100	1977	√	√			√	√	√	ACT, Battlemap, McCabe Instrumentation Tool, SLICE, Code-Breaker	CaseBridge						√
												BattlePlan	√	√	√	√		√
												START			√			
												McCabe OO Tool			√	√		
Program Analysers, Ltd.	+44 0635-528828	31	1987	√	√			√	√	√	LDRA Testbed							
Programming Environments, Inc.	(908) 918-0110	<20	1981		√				√	√	T	Runner				√		
Quality Engineering Software, Inc.	(203) 278-7252	<20	1991	√	√	√				√	QES/Manager	QES/Architect					√	
												QES/Qease					F	
												QES/Expert				F		
												QES/Programmer				F		
QualTrak Corp.	(408) 274-8867	12	1986	√	√			√	F	√	DDTs	RDDTs		√				
												QTET	F				F	
TA Consultancy Services, Ltd.	+44 252-711414	120	1974	√	√			√	√	√	MALPAS							
STARS Foundation Repository	(304) 594-9817	9	1989								TST							
Software Quality Assurance, Ltd.	+44 0277-374415	18	1984	√	√	√	√	F	F		T-PLAN							

Table 10-2 continued. Supplier Profiles

SUPPLIER NAME	SUPPLIER STATISTICS			SERVICES PROVIDED							Examined Tools	RELATED SUPPLIER TOOLS						
	Phone Number	Supplier Size	Established	Consultancy	Training	Test Planning	Test Performance	User Group	Newsletter	Hot-Line Support		Tool Name	Test Management	Problem Reporting	Static Analysis	Dynamic Analysis	Regression Analysis	CASE
Software Quality Automation	(800) 228-9922	15	1989	√						√	SQA:Manager	SQA:Robot					√	
Software Quality Engineering	(800) 423-8378	5	1985	√	√	√					SRE Toolkit							
Software Research, Inc.	(415) 957-1441	>25	1977	√	√	√	√	√	√	√	TCAT, S-TCAT, TCAT-PATH, TDGen, TSCOPE, METRIC	SMARTS	√				√	
												CAPBAK					√	
												EXDIFF					√	
												STATIC			√			
												TRACKER		F				
Software Systems Design	(714) 625-6147	<10	1985	√	√					√	TestGen, QualGen, GrafBrowse, ADADL Processor	AISLE Toolset			√			√
Testwell Oy	+358 31-165464	<5	1992							√	TBGEN, TCMON							
Verilog USA	(214) 241-6595	<20	1986	√	√			F		√	Logiscope	AGE/ASA			√	√		√
												AGE/GEODE			√	√		√
												DocBuilder						√

29. Ada-ASSURED

Ada-ASSURED is a set of multi-window, language-sensitive editors that analyze the compliance of code with the Ada Quality and Style Guidelines [SPC 1991] recommended by the Ada Joint Program Office and included in the BMDO Software Standards. Additionally, the toolset provides for program browsing and program reformatting.

29.1 Tool Overview

Ada-ASSURED was developed by GrammaTech and Loral Aerospace Corporation. The first version of this toolset, version 1.0, became available in November 1992. It runs under Unix and X11 Windows, with Motif and OpenWindows, on a variety of computers including the Sun SPARCstation, DECstation, HP 9000, and IBM RISC System/6000.

The evaluation was performed on Ada-ASSURED version 1.0 running on a Sun SPARCstation under Unix with OpenWindows. At the time of evaluation, the price for Ada-ASSURED started at $1,495.

Ada-ASSURED consists of three Ada language-sensitive editors:

- Ada Editor. Ensures syntactic correctness and consistent source code formatting.
- Ada Style-Enforcement Editor. Extends the Ada Editor with compliance checking against the SPC Ada Quality and Style Guidelines.
- ADL[1] Enforcement Editor. Extends the Ada Style-Enforcement Editor with language-sensitive editing of ADL and checking the consistency of Ada code with ADL constructs embedded as comments.

[1] ADL is an Ada program design and documentation language developed by the Loral Aerospace Corporation.

The examination focused on the style compliance and browsing functions of Ada-ASSURED, that is, the Ada Style-Enforcement Editor. Its capability as a language-sensitive editor was not examined, neither was its support for ADL.

The Ada Style-Enforcement Editor manipulates *objects* that are structured according to the grammatical rules of a particular language. The editor contains grammatical rules for several different languages including Ada and the language of hypertext cards. Each object is typed according to its linguistic category, called a *phylum*. The editor enforces the grammatical rules that describe permissible combinations of objects for the relevant phylum. Objects are browsed and edited in buffers, each associated with a particular phylum. Buffers themselves are associated with external files which may be files being edited or defined buffers that provide, for example, editing templates. Objects consist of phrases and while the structure contained in a buffer is a syntactically well-formed object of the phylum, it may contain phrases of unparsed text known as *text buffers*. It may also contain *placeholders*, that is, phrases that describe the phylum permitted or required at a given position.

Buffers are presented in textual *views* where each view is associated with a distinct collection of pretty printing rules. Views may differ radically, for example, including or excluding in-line error messages. Possible views include the following:

- Baseview. This is the default view and it contains a view of the object as defined in the enforcement parameters. Errors can appear in-line.
- Ada only. This is a view of the object as an Ada program with no style enforcement.
- Ada enforcement errors. This is a view containing enforcement error messages and terms that are considered placeholders.
- Ada-only errors. A view that displays errors detected in the Ada and placeholder terms.
- Ada enforcement indicators. Displays the enforcement parameters that are in effect, enforcement indicators, error messages, and placeholder terms.

Within a view, each phrase has a principal and other optional pretty printing schemes which govern how that phrase is displayed. Alternate pretty printing schemes allow, for example, elison where a phrase is displayed as " ... " and its detailed text is hidden.

A standard window has the typical title bar and menu bar. It also contains a message pane that presents error and advisory messages, an object pane displaying the object contained in a buffer, and context pane displaying context-sensitive status information and commands. Windows can be switched between views and buffers, and they can be duplicated and deleted.

Traditional window-type navigation such as using the insertion cursor, a locator such as a mouse, and scrolling are supported. Each object has a structural selection that identifies the current focus of structural edit operations and Ada-ASSURED also provides for navigation with structural selection. (Each buffer has only one structural selection regardless of the number of windows, the structural selection is updated in all windows displaying the buffer.) In this last type of navigation, the currently selected structure in the buffer can be moved relative to its current location using various treewalking regimes, for example, *preorder*, *reverse preorder*, *ascend-to-parent*, *forward-sibling*, *and struct-select-to-top*. These can be used to quickly find, for example, the next item in list. All commands that move the structural selection are parser-initiating. This means that text is (re)parsed if the text buffer has been modified since the last attempt to parse it, the target location is beyond the end of the text buffer, auto-parsing mode is enabled, or the current structural selection is a text buffer.

Parsing establishes the syntactic correctness of text buffers. As just mentioned, it can be invoked automatically by parsing-initiating commands when the auto-parsing mode is enabled, or it can be invoked manually. In addition to ensuring conformance to Ada syntax, parsing causes the Ada Style-Enforcement Editor to analyze Ada code with respect to the SPC Ada Quality and Style Guidelines. The automatic formatting provided by the language-sensitive editor means that some of these guidelines are automatically enforced; including, for example, horizontal spacing and indentation. Otherwise, the user can set enforcement parameters to enable or disable the enforcement of individual style guidelines. When a parameter is changed, the user can choose to apply it immediately to the current buffer.

Ada-ASSURED provides two levels of enforcement: indicators and errors. In general, indicators are used to signify guideline violations that are not considered critical; they can be treated as warnings and only appear in a special indicator view. Errors, on the other hand, denote more serious guideline violations that should not be ignored and they shown within the code as in-line error messages contained in comments. In some cases, when a guideline is violated, Ada-ASSURED offers an automatic transformation to bring the code into compliance.

A script interpreter provides the ability to invoke Ada-ASSURED editors, either interactively or in batch mode, with a script of commands. This capability allows the user to provide new compound editing actions that extend the functionality of the editor, to invoke an initialization script on start up to customize editor operation, and to define their own templates which can be used to replace appropriate placeholders. It also allows editors to serve as batch tools such as pretty printers and program analyzers. (An editor running in batch mode does not start up any windows, the result of interpreting a script is printed to a special buffer **transcript**.) Scripts are written in a language called Dynamic SSL and consist of function definitions that may invoke built-in library functions and commands. INT, REAL, BOOL, CHAR, and TOK are provided as primitive phyla and all editing functions are available as built-in primitive functions. A special script editor with a complete set of parsing rules and transformations is available to support their creation and maintenance. This editor also provides special commands that allow scripts to be interpreted from within the editor.

29.2 Observations

Ease of use. Ada-ASSURED uses on-line hypertext to provide context-sensitive help and a cross-reference to the Ada Language Reference Manual. Additional aid for the user is provided through facilities such as a file-and-directory browser that can be used with commands that read and write files to assist in locating file names of interest.

The hypertext help can be extended by preparing additional hypertext cards and linking them into the basic hypertext web. The standard X resource interface can be customized to affect, for example, key bindings and type face. Menus are available to adjust window layout and features such as wordwrap. There are several levels of customization available for the editors. Editor style can be adjusted, for example, via the style templates associated with each editor. Style is defined in terms of font, size, slant, weight, and color. Styles are derived by successively applying templates to the default style, itself generated by applying the default template to a *root* style that is generated when the editor is invoked. The following styles are provided with Ada-ASSURED and can be customized: Background, Comment, Error, Indicator, Keyword, and Placeholder. Dynamic SSL provides another method for customizing an editor through interpreted scripts. Alternatively, the user can exploit the fact that Ada-ASSURED is itself generated by GrammaTech's Synthesizer Generator from a high-level, rule-based description. By modifying this description the language-sensitive editor generator allows rapid production of a customized editor.

Documentation and user support. A reference manual is available on-line as a web of hypertext cards. An on-line tutorial is provided that uses these cards.

Ada restrictions. None.

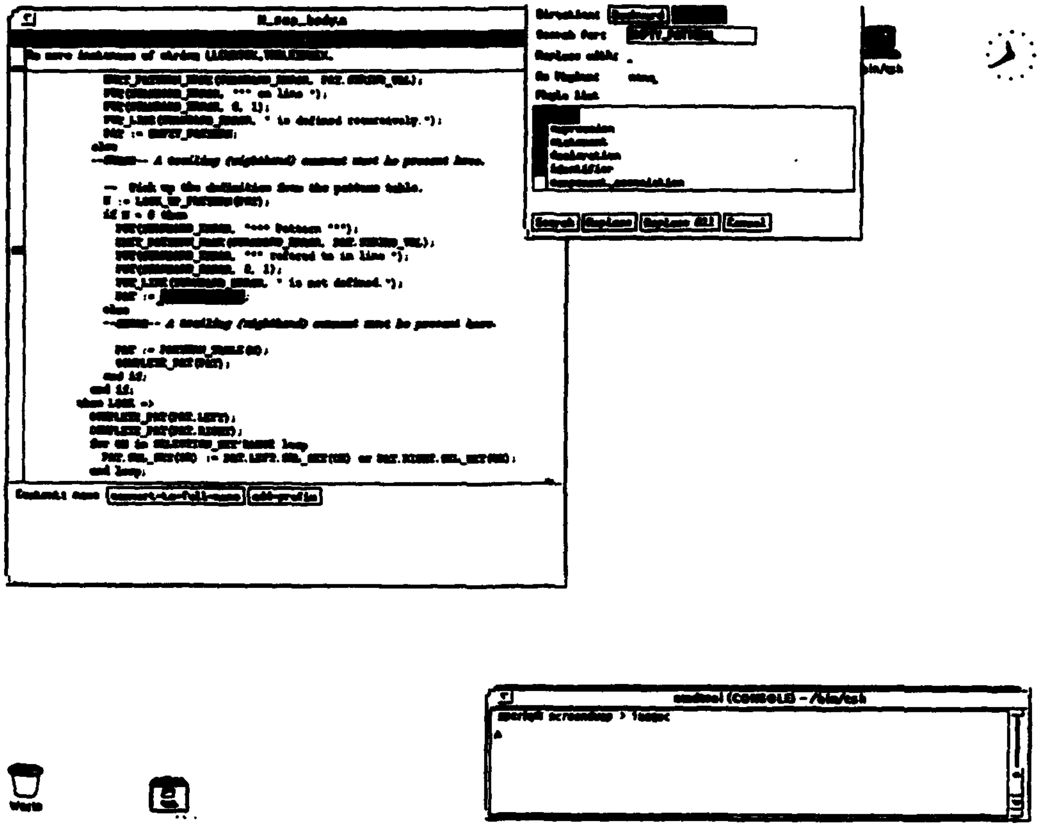
Problems encountered. The only problem encountered was in running one of the sample Dynamic SSL scripts that were provided. This script contained an error that needed correction. Otherwise, Ada-ASSURED performed as described in the documentation.

29.3 Planned Additions

Subsequent to this review, GrammaTech have released Ada-ASSURED version 1.2 This new version provides facilities for integration with any compiler. Future versions of Ada-ASSURED will provide the capability for semantic browsing and integration with tool managers.

29.4 Sample Outputs

Figures 29-1 through 29-6 provide sample outputs from Ada-ASSURED.



```
No more instances of string [illegible].
            [illegible](..., PAT.[illegible]_VAL);
            PUT([illegible]_ERROR, "** on line ");
            PUT([illegible]_ERROR, 0, 1);
            PUT_LINE([illegible]_ERROR, " is defined recursively.");
            PAT := EMPTY_PATTERN;
          else
          --[illegible]-- A trailing (righthand) comment must be present here.

            -- Pick up the definition from the pattern table.
            N := LOOK_UP_PATTERN(PAT);
            if N = 0 then
              PUT([illegible]_ERROR, "*** Pattern ***");
              [illegible](..., PAT.STRING_VAL);
              PUT([illegible]_ERROR, "** referred to in line ");
              PUT([illegible]_ERROR, 0, 1);
              PUT_LINE([illegible]_ERROR, " is not defined.");
              PAT := [illegible];
            else
            --[illegible]-- A trailing (righthand) comment must be present here.

              PAT := PATTERN_TABLE(N);
              COMPLETE_PAT(PAT);
            end if;
          end if;
        when LOOK =>
          COMPLETE_PAT(PAT.LEFT);
          COMPLETE_PAT(PAT.RIGHT);
          for [illegible] in [illegible]'range loop
            PAT.[illegible](..) := PAT.LEFT.[illegible](..) or PAT.RIGHT.[illegible](..);
          end loop;
```

Direction: Backward [Forward]

Search for: EMPTY_PATTERN

Replace with:

On Replace: [illegible]

Phyla list:

```
expression
statement
declaration
identifier
component_association
```

Figure 29-1. Ada-ASSURED Structural Search

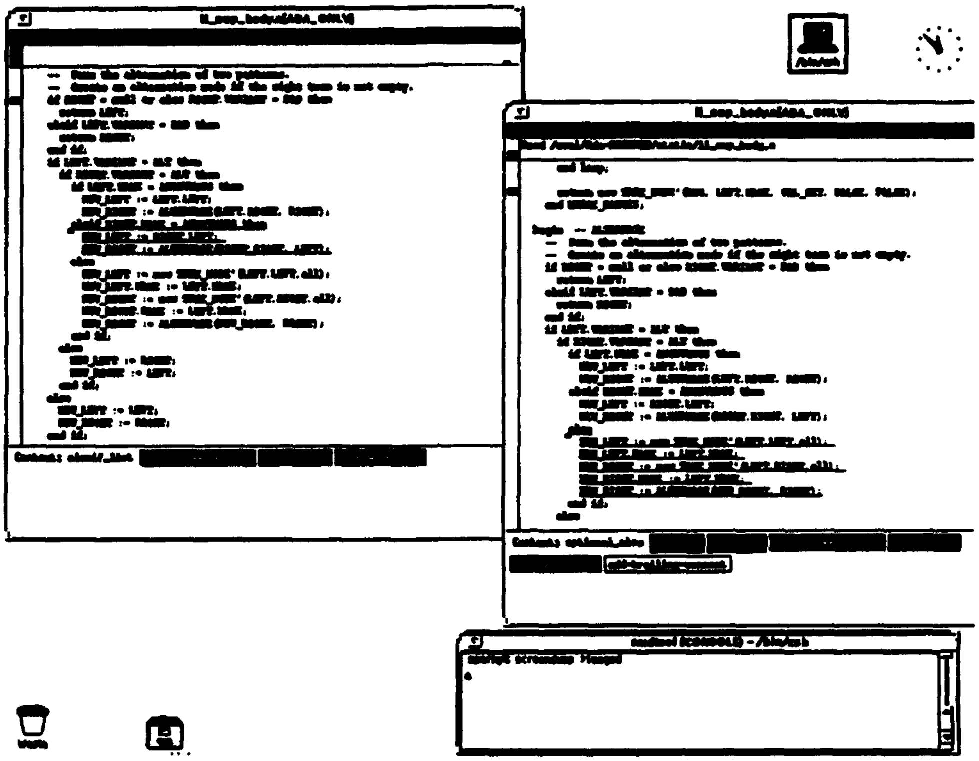
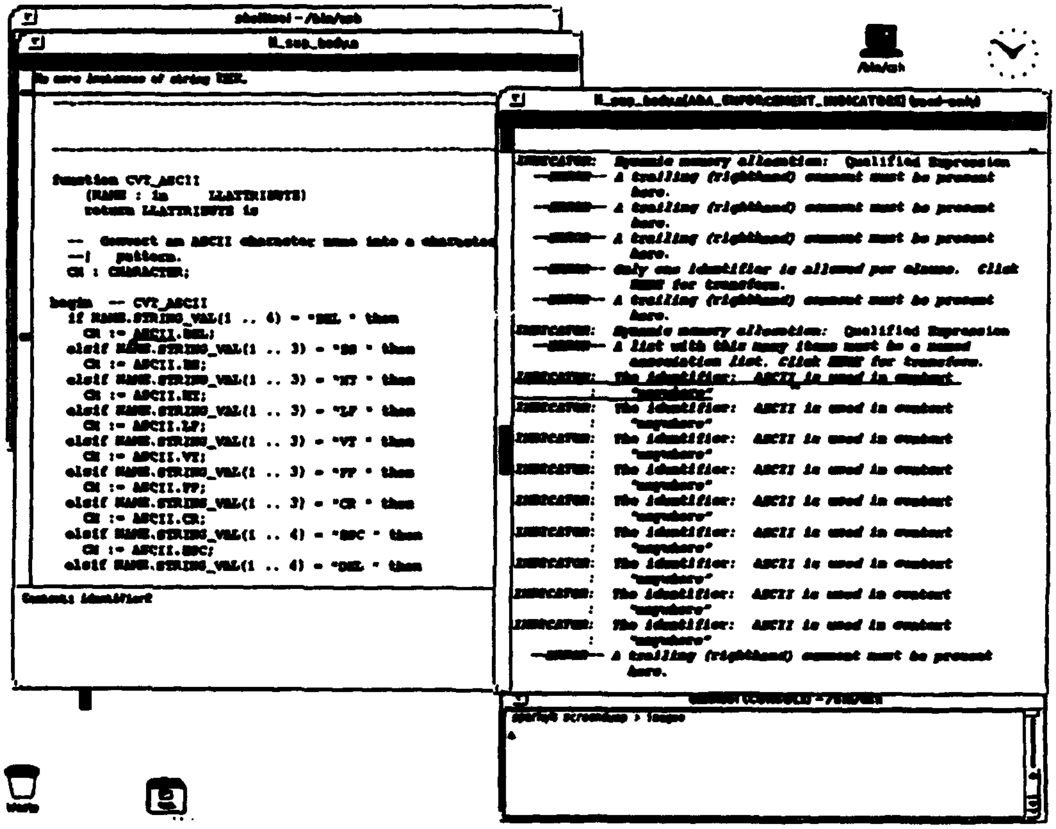


Figure 29-2. Ada-ASSURED Navigation by Structural Selection



No more instances of string DEL.

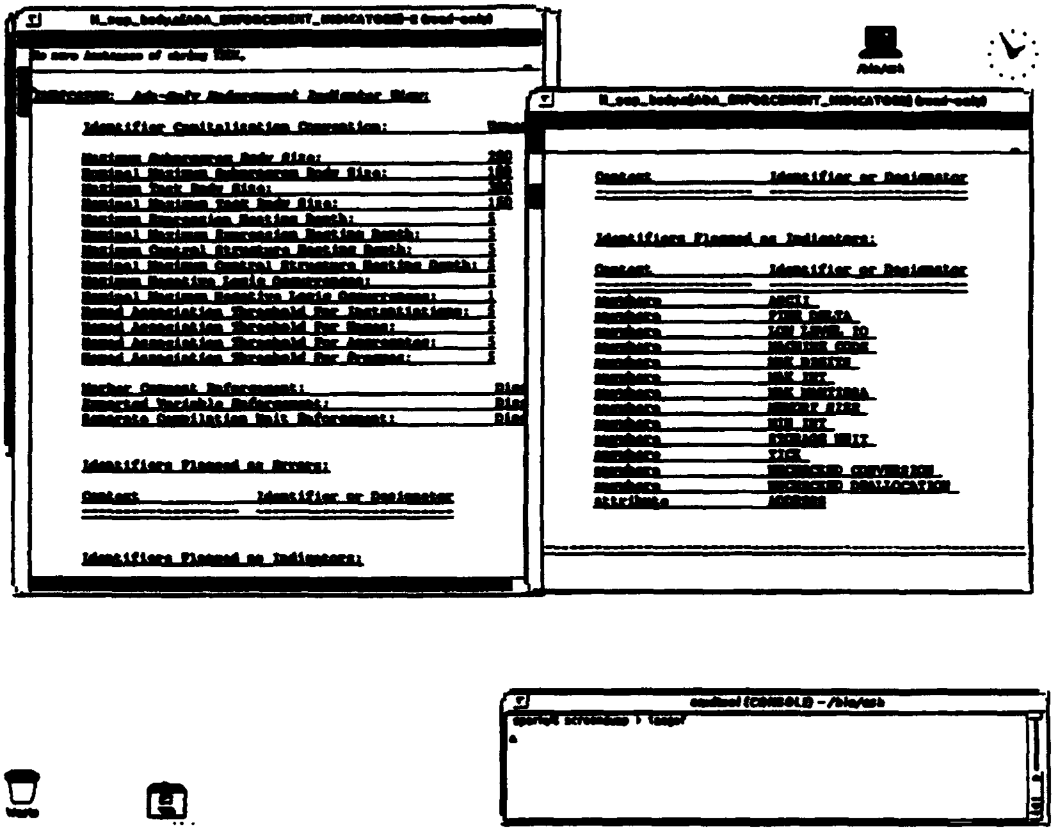
```
function CVT_ASCII
    (NAME : in     LLATTRIBUTE)
    return LLATTRIBUTE is

  --  Convert an ASCII character name into a character
  --|   pattern.
  CH : CHARACTER;

begin  -- CVT_ASCII
  if NAME.STRING_VAL(1 .. 4) = "BEL " then
    CH := ASCII.BEL;
  elsif NAME.STRING_VAL(1 .. 3) = "BS " then
    CH := ASCII.BS;
  elsif NAME.STRING_VAL(1 .. 3) = "HT " then
    CH := ASCII.HT;
  elsif NAME.STRING_VAL(1 .. 3) = "LF " then
    CH := ASCII.LF;
  elsif NAME.STRING_VAL(1 .. 3) = "VT " then
    CH := ASCII.VT;
  elsif NAME.STRING_VAL(1 .. 3) = "FF " then
    CH := ASCII.FF;
  elsif NAME.STRING_VAL(1 .. 3) = "CR " then
    CH := ASCII.CR;
  elsif NAME.STRING_VAL(1 .. 4) = "ESC " then
    CH := ASCII.ESC;
  elsif NAME.STRING_VAL(1 .. 4) = "DEL " then
```

Context: Identifier?

```
INDICATOR:  Dynamic memory allocation:  Qualified Expression
   —ERROR— A trailing (righthand) comment must be present
             here.
   —ERROR— A trailing (righthand) comment must be present
             here.
   —ERROR— A trailing (righthand) comment must be present
             here.
   —ERROR— Only one identifier is allowed per clause.  Click
             MENU for transform.
   —ERROR— A trailing (righthand) comment must be present
             here.
INDICATOR:  Dynamic memory allocation:  Qualified Expression
   —ERROR— A list with this many items must be a named
             association list. Click MENU for transform.
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
INDICATOR:  The identifier:  ASCII is used in context
          :    "anywhere"
   —ERROR— A trailing (righthand) comment must be present
             here.
```

```
[illegible]
sparky% screendump > image
```

Figure 29-3. Ada-ASSURED Indicators vs. Errors



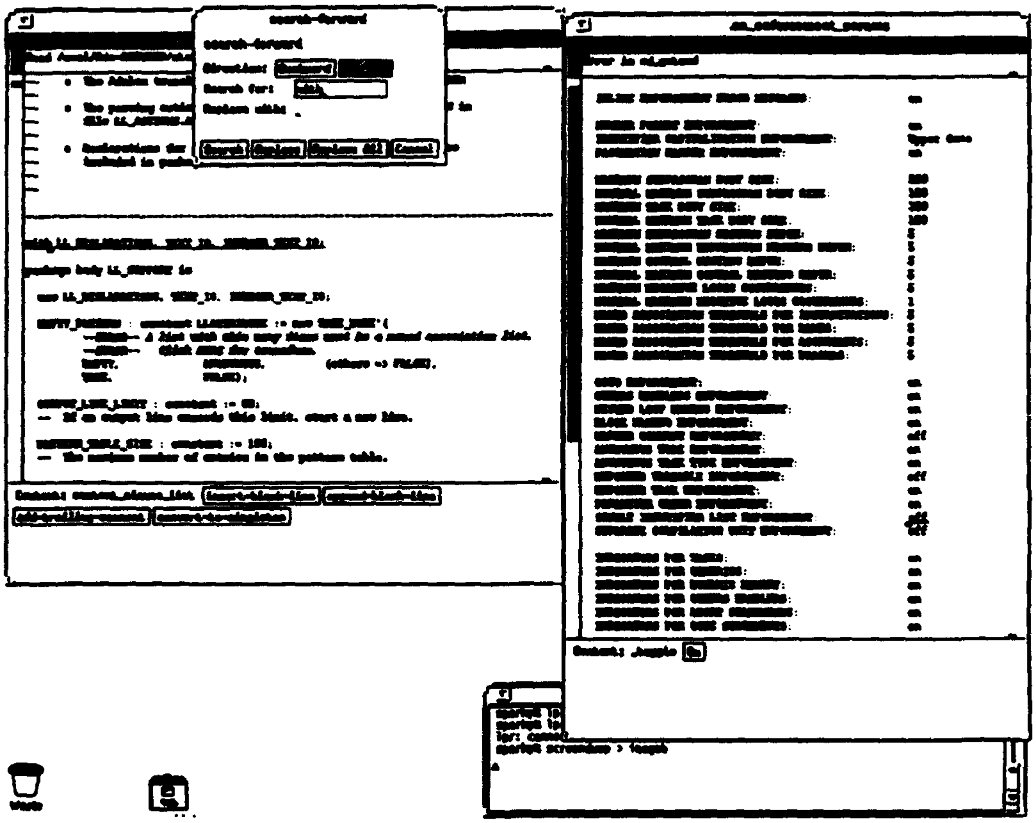


Figure 29-5. Ada-ASSURED Enforcement Parameters

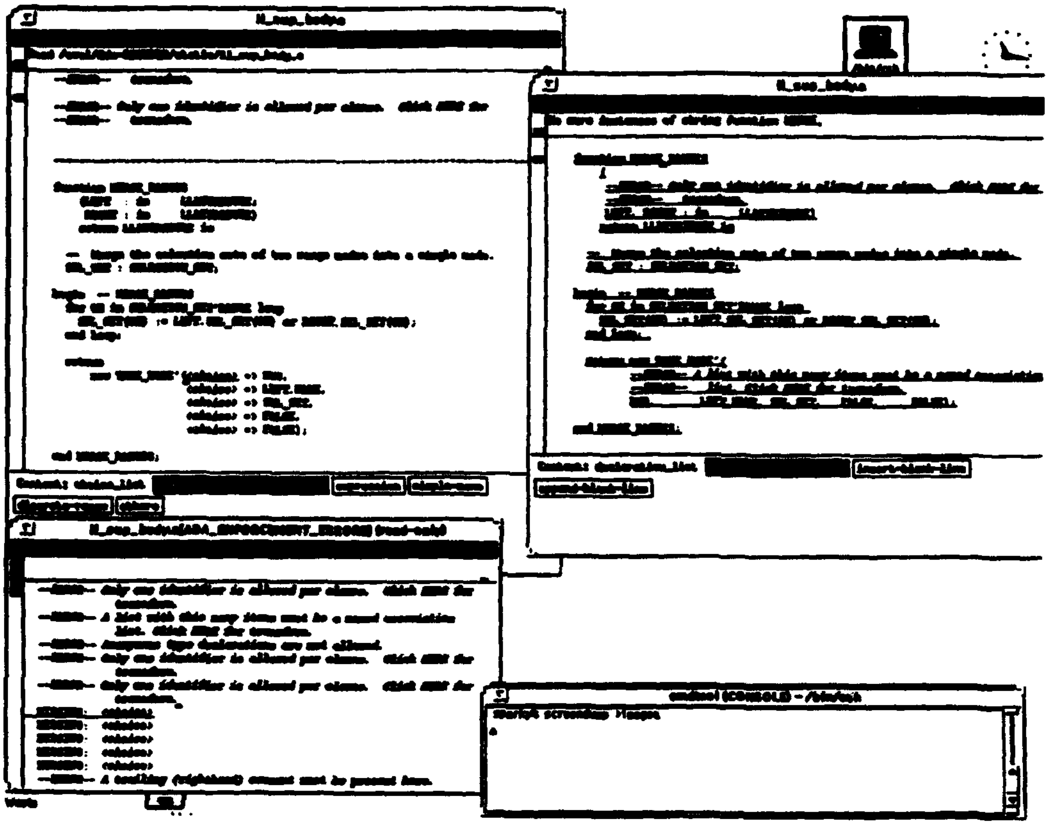


Figure 29-6. Ada-ASSURED Automatic Error Correction

30. AdaTEST

AdaTEST consists of three components: the AdaTEST Harness (ATH), the AdaTEST Analyser (ATA), and the AdaTEST Instrumentor (ATI). It supports unit testing and bottom-up testing of Ada code. ATH is a library of Ada packages that provide facilities to write a test script that supports white box testing of Ada program units. It passes data to the unit under test and allows the user to check the results of the unit, including the correct propagation of exceptions. ATH also supports simulating the interaction between the unit under test and other (perhaps undeveloped) units and, finally, verifying that run-time performance requirements are met. ATI provides static analysis to calculate code complexity of the unit under test. It also instruments the code to provide statement, decision, boolean expression, exception, and call coverage. Finally, ATA is another library of Ada packages that allows testing the instrumented unit in the same way as ATH packages, and additionally reporting on the static and coverage analyses. ATA also supports tracing the path of execution through the unit under test.

AdaTEST was developed for use in the high-integrity/safety-critical development arena. Accordingly, ATH, the core of AdaTEST, was developed as a safety-critical development in its own right, using the same standard specified for use in the Eurojet project (part of the European Fighter Aircraft project). The remainder of AdaTEST was developed to IPL's usual development standards, including ISO 19001.

30.1 Tool Overview

AdaTEST was developed by Information Processing Limited in Bath, England. IPL also provide training, consultancy, and hot-line support to tool users. They have recently established a relationship with a U.S. company, Texel & Co., who will market and support AdaTEST in this country. AdaTEST was first marketed in 1991 and is now used at more than 15 sites internationally. It is machine and operating system independent, relying only on the available Ada compiler. The examination was performed on AdaTEST Harness (ATH) version 2.3, AdaTEST Analysis (ATA) version 2.1, and AdaTEST Instrumentor

(ATI) version 2.1. These ran on a Sun-3 workstation under a Verdix Ada Development System. At that time, the price for AdaTEST started at £9,500.

30.1.1 ATH Overview

The user begins with AdaTEST by using the library of packages that constitute ATH to construct a test script. This test script will act as a main procedure and invoke the unit under test. Consequently, a test script is compiled and linked as a normal Ada main program, and changes to the test script only require its recompilation and the relinking of the program; the software under test is unaffected.

ATH provides a number of directives that are used in the test script to control the invocation of the unit under test and to check its results. The structure of the test script, and placement of directives, is governed by two state machines: the script state machine and the timer state machine. If the user causes a directive to be called from an inappropriate state, a script error is generated and the test fails. (A set of diagnostic error messages helps the user to determine the cause of a script error, and limited error recovery is provided by adopting the correct state after a script error has been given. Of course, side effects from the error may prevent complete recovery.)

The user is provided with a template that aids in test script construction. A test script starts by setting the context in terms of *with*ing and *use*ing needed packages. Then the script is opened and any necessary test data is declared, followed by the instantiation of necessary check procedures. A series of test cases specify the data to be passed to the unit under test, the data expected to be returned by the unit, and the various checks that should be made to check correct unit operation. After the main body of the test script is closed, the stub section provides the definition of any stubs that are to be simulated. The basic operation of this part of AdaTEST can be illustrated be looking at some of the major directives that are used in the main body and stub section of a test script.

Basic directives are used to declare the start and end of a test script, and start and end of each individual test case. Within each test case, the EXECUTE directive precedes the actual invocation of the unit under test. It allows the user to identify the unit, the expected sequence of stub calls, if any, made by the unit, and whether an exception is expected to be propagated. A companion directive DONE follows the invocation of the unit under test and verifies that the number of stub calls and any exception propagation is as stated in the preceding EXECUTE. Next a series of checks can be made to verify the test case outputs..

CHECK is used to verify that an object of a predefined Ada type contains the expected value. To allow more flexibility, ATH also provides a set of generic checking routines that the user can use to check the results for user-defined integer, enumeration, floating, and fixed types and, with the exception of limited private types, any other user-defined types. A check directive for comparing raw memory is also provided. In addition to checking the value of data against expected data, ATH provides for checking of events. Here expected external physical events can be checked by requesting the operator to confirm that a particular event has occurred. Also the exceptions propagated by the unit under test can be examined. The CORRECT_EXCEPTION and ILLEGAL_EXCEPTION directives allow the test to indicate whether an exception has been propagated correctly, or unexpectedly.

Timing analysis is supported by special directives to reset, start, and stop the timer and check timer results. Since the processing involved in stub calls can have an effect on the execution of the unit under test, it is recommended that a test script should not use simulation when timing analysis is being performed. For a test script to be portable between host and target environments, timing checks are specific to the environment in which the test is executed. Timing directives are ignored in the host environment and this type of analysis is only performed in the target environment.

AdaTEST's stub simulation supports testing units in isolation, where interaction with other units is simulated. (In bottom-up testing, stub simulation can be used to replace those units not yet available.) The user can verify the order of calls to simulated units, and verify the values of in and in-out parameters passed to simulated units. The user is also provided with further control in the test process by the ability to initialize the values of in-out and out parameters, function return values, and other data. The sequence of expected calls to simulated units is usually given in the EXECUTE directive included in each test case to allow the user to specify unique processing for each separate stub invocation. The simulated unit(s) themselves are defined in the test script after the main body of test cases. Again, special script directives are provided. The directives START_STUB and END_STUB are self-explanatory. CALL_REF is used to identify the current stub reference from the list given in EXECUTE directive. Since the list of expected calls in an EXECUTE directive can contain loops, the directive CALL_LOOP is available to identify the current value of a loop repetition count and another directive, the ILLEGAL_CALL_REF directive, can be used to indicate that a simulated unit has been called out of sequence. This use of stub simulation is facilitated if all units in a system are declared as separate, thus obviating the need to write hybrid package bodies containing both simulated and real units.

At the end of a test run, a table of test results is produced. This indicates whether each test has passed or failed and gives an overall pass/fail summary of the test run. In addition, detailed results are produced. Tests may be executed unchanged in both host and target environments (with the exception of timing analysis, as stated above). In a host environment the detailed test output generated by a test run is placed in a results file for later examination. On a target, if no file system is available, output is directed to the standard output device.

Since a large number of test cases can make a test script unwieldy, ATH supports table-driven testing. Here the initial conditions and expected results for each test case are stored in a table. This allows the test script to simply loop, pulling the needed data for each test case from the table. Of course, although the table can be extended to allow for different sequences of stub calls and different handling of exceptions, this approach assumes a large degree of commonality in the processing required for each test case. However, in those cases where a target system has limited memory resources, table-driven testing does allow reducing the size of the executable program.

Tasks are tested in much the same way as sequential units. While sequential units are implicitly synchronized because they are executed from within the EXECUTE-DONE block, tasks may run concurrently with the test harness. If a task has entry points, then the entry point is called after the EXECUTE and a delay statement precedes the DONE to allow time for the task to call any stubs and update any global data. If it is not possible to synchronize the task using an entry point, or the entry point occurs too late (for example, after a stub has been called), special action is called for. In this case, the CONCURRENT mode of ATH must be used. In this mode, directives are locked so that only one may execute at a time and stubs are executed sequentially. Essentially, ATH synchronizes START_STUB with EXECUTE so that stub calls made by the unit under test will not proceed until the test script is within the EXECUTE-DONE block. Again, a delay statement may be necessary. Similarly END_STUB is synchronized with DONE. After DONE has executed, any subsequent stub calls are suspended until the next EXECUTE. Some workarounds are provided for cases such as a task reading global data before waiting on a task entry or calling a stub.

CONCURRENT mode is also required for multiple tasks and, although this is not stated in the manual, for coverage analysis of a task. The ATH tasking mode (SEQUENTIAL or CONCURRENT) is determined when the test script and software under test are linked to form an executable image. The manner of setting the mode is dependent on the compilation environment. Since the CONCURRENT mode is intrusive, it may affect the timing and ordering of events that occur across multiple tasks; this may result in deadlocks where the program hangs.

30.1.2 ATI Overview

Unlike ATH and ATA, ATI is an executable program. As such it offers a command line interface for instrumenting subprogram bodies, task bodies, and subunits when they appear as library units or in the declaration part of other units of nested block statements. (Code embedded in a package body is not instrumented.) Instrumentation is performed on a single file at a time.

The command line interface offers a number of options that allow, for example, limiting the types of coverage that the unit is instrumented for, switching off task body instrumentation, and requesting static analysis results to be produced in a machine-readable, comma-separated variable format suitable for processing by third-party products such as spreadsheet packages. It is also possible to limit the instrumentation performed to, for example, a maximum number of coverage points, or the maximum limit for the number of bytes used to store package and program unit names. By default, the subject file is instrumented for all coverage types, that is, statement, decision, boolean expression, exception, and call coverage.

Two files are produced. One of these includes the instrumented source code, which is ready to be compiled in the usual way. The second file contains an annotated listing of the source code and the results of the static analysis. Static metrics are calculated on a per-unit basis. They include simple counting metrics such as the total number of Ada source code lines, the total number of dynamic stack allocations within a unit, and the total number of for loops. Complexity metrics include Halstead metrics, and Hansens's Cyclomatic number and operator counts which are modifications of McCabe's and Myers' complexity metrics.

30.1.3 ATA Overview

After it has been instrumented, a program unit is ready for coverage analysis. The user converts the ATH test script used for testing the program unit into an ATA test script by including ATA directives that cause the capture, checking, and reporting of coverage analysis data. The initial ATA directive used is one to initialize the analysis, then the test cases are surrounded by directives to start and stop coverage analysis. A CHECK_ANAL-

YSIS directive is used to check that, for a given unit, a particular coverage metric or static metric lies within a specified range. Alternatively, the user can use the GET_ANALYSIS directive to yield the value of a particular metric and this data can be used in, for example, the calculation of user-defined metrics. CHECK_CALLS checks that all members of a given list of units have been executed. The directives REPORT_UNIT and REPORT_ANALYSIS allow writing the results of static analysis to the results file either unit by unit or metric by metric. A trace directive is provided to allow the user to request tracing at either the unit, decision, or statement level.

START_SCRIPT_IMPORT and END_SCRIPT_EXPORT are the most important of the remaining directives. These are used as alternatives to the usual START_SCRIPT and END_SCRIPT to enable cumulative coverage reporting. As its name suggests, START_SCRIPT_IMPORT imports analysis data and pass/fail results exported from a previous run, new results are then appended to this data and then, using END_SCRIPT_EXPORT, can be exported to a set of files for later import by another test script.

Once the ATA test script is prepared, all relevant code is compiled and linked as usual. (As before, any changes to the test script only require recompiling the script itself and relinking the program.) When the resulting program is run, results of the analysis and tracing are integrated with ATH test results to produce an overall ATH/ATA test pass/fail. A summary of the pass/fail results is written to the screen, while the full analysis and, if requested, trace results are written to the results file.

ATA operates in one of two modes, either ANALYSIS or NON_ANALYSIS. In the ANALYSIS mode, ATA operates as described above., Otherwise ATA ignores all directives except the START_SCRIPT_IMPORT and END_SCRIPT_EXPORT; it will not permit the execution of instrumented code in this mode. This is intended to allow the use of the same test script for both host and target machine testing. It also allows timing analysis to be restricted from being performed on instrumented code.

30.2 Observations

Ease of use. For ATH and ATA, knowledge of Ada is all that is required. ATI uses a simple command-line interface that is easy to use.

Documentation and user support. The documentation as provided was clear and easy to follow. However, it serves as a reference manual rather than a user guide and additional examples would be helpful. IPL provided excellent support, answering all questions quickly and fully.

Instrumentation overhead. The majority of the coverage analyzers examined in the course of this work are intended for instrumentation of entire programs, or significant portions of a program. ATA differs in that it is primarily intended to monitor the coverage achieved in testing a single program unit or, with appropriate use of test histories, those program units examined to date in the course of bottom-up testing. Consequently, the following figures should not be directly compared with those given for other coverage analysis tools.

For the function LLFIND, instrumentation for boolean expression, decision, statement, and exception coverage gave an increase in the size of source code from 810 to 4,924 blocks. For package BUFFER_INPUT_MSGS, containing the BUFFER task, instrumentation gave an increase from 2,118 to 10,811 blocks of source code.

Ada restrictions. AdaTEST supports full Ada. However, ATA does not instrument boolean expressions that contain a relational operator with one or more operands that are also boolean expressions.

Problems encountered. No problems were encountered during use of AdaTEST. It operated exactly as described in the manual.

30.3 Planned Additions

IPL is working on extending AdaTEST to support data flow testing. IPL is also working on links to two major CASE tools which will allow automatic test script generation from the case design level.

30.4 Sample Outputs

Figures 30-1 through 30-13 provide sample outputs from AdaTEST on the Lexical Analyzer Generator example, and Figures 30-14 through 30-23 provide sample outputs for the Real-Time Temperature Monitor example.

```
separate (LLFIND_TEST_PKG)

function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
   --  Find item in symbol table -- return index or 0 if not found.
   --  Assumes symbol table is sorted in ascending order.
   LOW, MIDPOINT, HIGH: INTEGER;
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      MIDPOINT := (HIGH + LOW) / 2;
      if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
         HIGH := MIDPOINT;
      elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
            return( MIDPOINT );
         else
            return( 0 );
         end if;
      else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
         LOW := MIDPOINT + 1;
      end if;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;
```

Figure 30-1. AdaTEST Listing of Function LLFIND

```
with  LL_DECLARATIONS, TEXT_IO;
package LLFIND_TEST_PKG is
use  LL_DECLARATIONS, TEXT_IO;

type  LLSYMTABENTRY  is      -- for symbol table entries
   record
      KEY: LLSTRINGS;    -- literal string or group identifier
      KIND: LLSTYLE;     -- literal or group
   end record;

LLSYMBOLTABLE: array ( 1 .. LLTABLESIZE ) of LLSYMTABENTRY;
                             -- the symbol table for literal terms

function LLFIND ( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER;
procedure READGRAM;

end LLFIND_TEST_PKG;


package body LLFIND_TEST_PKG is

    function LLFIND ( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
                              separate;

    procedure  READGRAM  is      -- read grammar from disk
       CH: CHARACTER;
       LLGRAM: FILE_TYPE;      -- where grammar is stored
    begin   -- READGRAM
       OPEN( LLGRAM, IN_FILE, "TABLE" );
       -- read in symbol tables
       for  I  in  1 .. LLTABLESIZE  loop
          for  J  in  1 .. LLSTRINGLENGTH  loop
             GET( LLGRAM, LLSYMBOLTABLE(I).KEY(J) );
          end loop;
          GET( LLGRAM, CH );
          SKIP_LINE( LLGRAM );
          if  CH = 'g'  then
             LLSYMBOLTABLE(I).KIND := GROUP;
          else   -- assume ch = 1
             LLSYMBOLTABLE(I).KIND := LITERAL;
          end if;
       end loop;
       CLOSE( LLGRAM );
    end  READGRAM;

end LLFIND_TEST_PKG;
```

Figure 30-2. AdaTEST Making LLFIND into a Separate Unit

```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- CONTEXT
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with LLFIND_TEST_PKG;
use LLFIND_TEST_PKG;

-- with any Referenced Packages:
with LL_DECLARATIONS;
use LL_DECLARATIONS;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
with ADATEST_HARNESS_TIMING_ANALYSIS ;
use  ADATEST_HARNESS_TIMING_ANALYSIS ;

-- Test script may require the use of other ATH packages, e.g :
-- with ADATEST_HARNESS_GENERIC_CHECKS ;
-- use  ADATEST_HARNESS_GENERIC_CHECKS ;
---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- OPEN SCRIPT
---------------------------------------------------------------------------
---------------------------------------------------------------------------
procedure TEST_LLFIND is

-- Instantiations of Generic_Checks if needed:

type LLSTYLE is (LITERAL, NONTERMINAL, GROUP, ACTION, PATCH) ;

-- Declare work variables if needed:
  ITEM : LLSTRINGS;
  WHICH : LLSTYLE;
  RESULT : INTEGER;

begin
     START_SCRIPT ( "TEST_LLFIND" ) ;

          -- Read in the table to initialise LLSYMBOLTABLE
          READGRAM;

-- Timer reset here for timing analysis later [illegible]st script
     RESET_TIMER ;
------------------------------------------------------------------------------
------------------------------------------------------------------------------
-- TEST PATHS (Repeat as required)
------------------------------------------------------------------------------
--      TEST CASE 1
------------------------------------------------------------------------------
       START_TEST ( 1 ) ;
          -- COMMENT ( "ITEM => #, 1st item in table" ) ;
          -- COMMENT ( "WHICH => 1" ) ;
          -- COMMENT ( "RESULT => 1" ) ;

-- No initial values to set

-- Introduce call to Unit_under_Test with Call_List and Is_Exception_Expected:
```

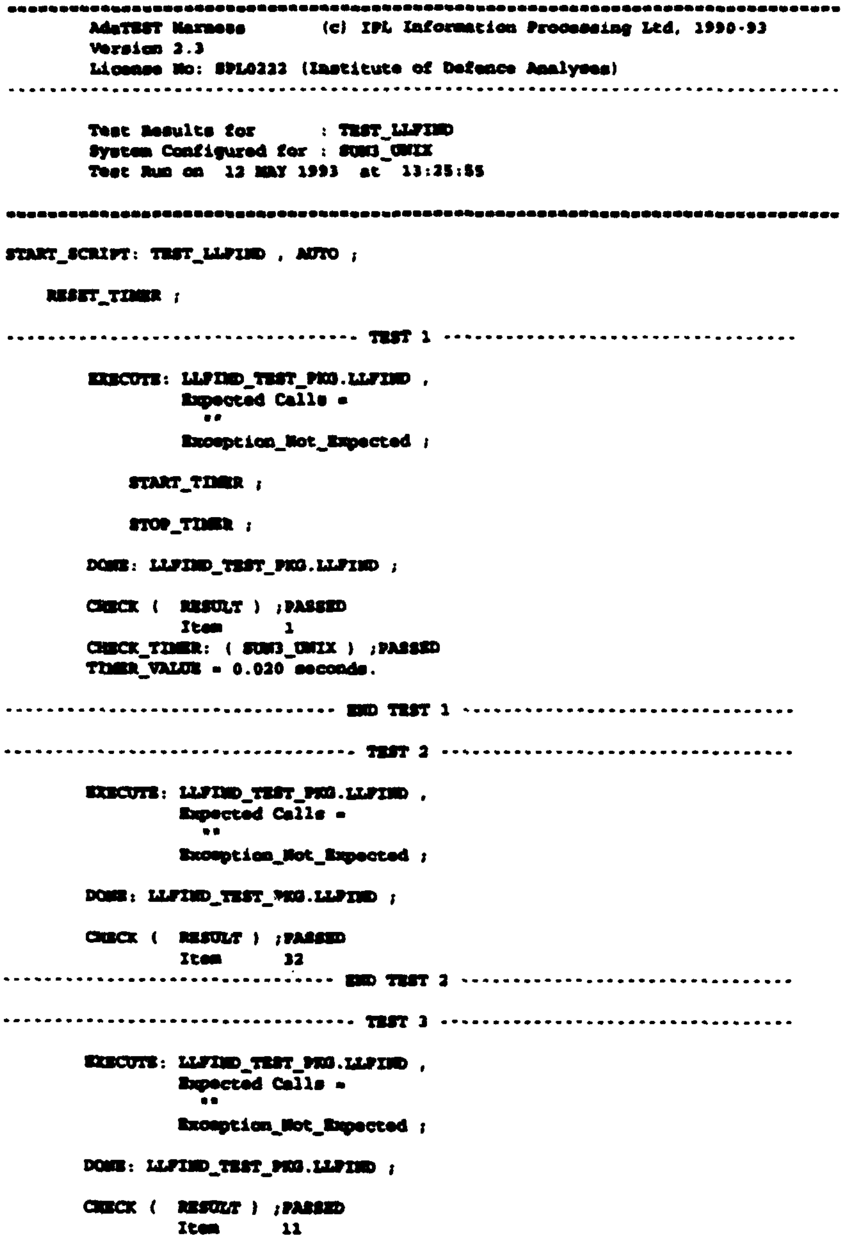
Figure 30-3. ATH Test Script for Testing LLFIND with Timing Analysis

```
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED ) ;

           begin
             START_TIMER ;
-- Make call to Unit_under_Test with appropriate value(s) for In parameters
-- and work variables for Out parameters and returned values:
             RESULT := LLFIND(ITEM => "#                   ", WHICH => LITERAL);
             STOP_TIMER ;

-- Exception handler(s):
           exception

        -- If test case requires, provide a Correct_Exception handler, e.g.
        -- when PACKAGE_UNDER_TEST.EXPECTED_EXCEPTION =>
        --      CORRECT_EXCEPTION ("EXPECTED_EXCEPTION") ;
        -- Always provide an ILLEGAL_EXCEPTION handler for at least 'others':
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;

           end ;
           DONE ;

-- Check the results of executing the Unit_under_test:
           CHECK ( "RESULT", RESULT, 1) ;
           CHECK_TIMER ("SUN3_UNIX", 0.00, 0.05) ;

         END_TEST ;
-------------------------------------------------------------------------------
--      TEST CASE 2
-------------------------------------------------------------------------------
       START_TEST ( 2 ) ;
          -- COMMENT ( "ITEM => }, last item in table" ) ;
          -- COMMENT ( "WHICH => 1" ) ;
          -- COMMENT ( "RESULT => 32" ) ;
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED ) ;

           begin
             RESULT := LLFIND(ITEM => "}                   ", WHICH => LITERAL);
           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;
           DONE ;
           CHECK ( "RESULT", RESULT, 32) ;

         END_TEST ;
-------------------------------------------------------------------------------
--      TEST CASE 3
-------------------------------------------------------------------------------
       START_TEST ( 3 ) ;
          -- COMMENT ( "ITEM => CharLit, some random entry" ) ;
          -- COMMENT ( "WHICH => GROUP" ) ;
          -- COMMENT ( "RESULT => 11" ) ;
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED ) ;

           begin
             RESULT := LLFIND(ITEM => "CharLit             ", WHICH => GROUP);
           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
```

Figure 30-3 continued: ATH Test Script for Testing LLFIND with Timing Analysis

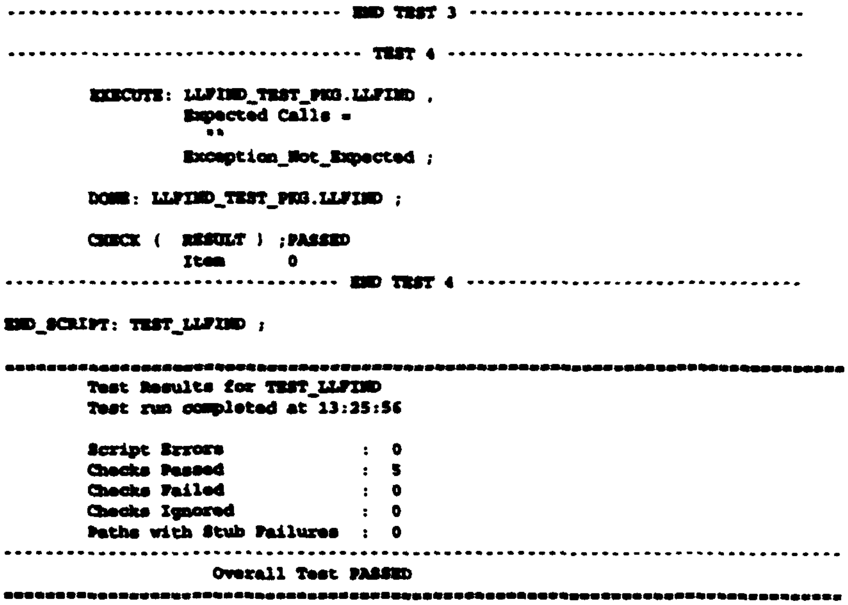
```
          end ;
          DONE ;
          CHECK ( "RESULT", RESULT, 11) ;

        END_TEST ;
------------------------------------------------------------------------------
--      TEST CASE 4
------------------------------------------------------------------------------
       START_TEST ( 4 ) ;
          -- COMMENT ( "ITEM => rubbish, item not present in table" ) ;
          -- COMMENT ( "WHICH => LITERAL, valid LLSTYLE" ) ;
          -- COMMENT ( "RESULT => 0" ) ;
          EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED ) ;

          begin
            RESULT := LLFIND(ITEM => "rubbish              ", WHICH => PATCH);
          exception
          when others =>
              ILLEGAL_EXCEPTION ( "Others" ) ;
          end ;
          DONE ;
          CHECK ( "RESULT", RESULT, 0) ;

        END_TEST ;
-----------------------------------------------------------------------------
------------------------------------------------------------------------------
-- CLOSE SCRIPT
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
        END_SCRIPT ;
end TEST_LLFIND;
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- No Stub Simulation section
------------------------------------------------------------------------------
-------------------------------------------------------------------------------
```

Figure 30-3 continued: ATH Tes. Script for Testing LLFIND with Timing Analysis



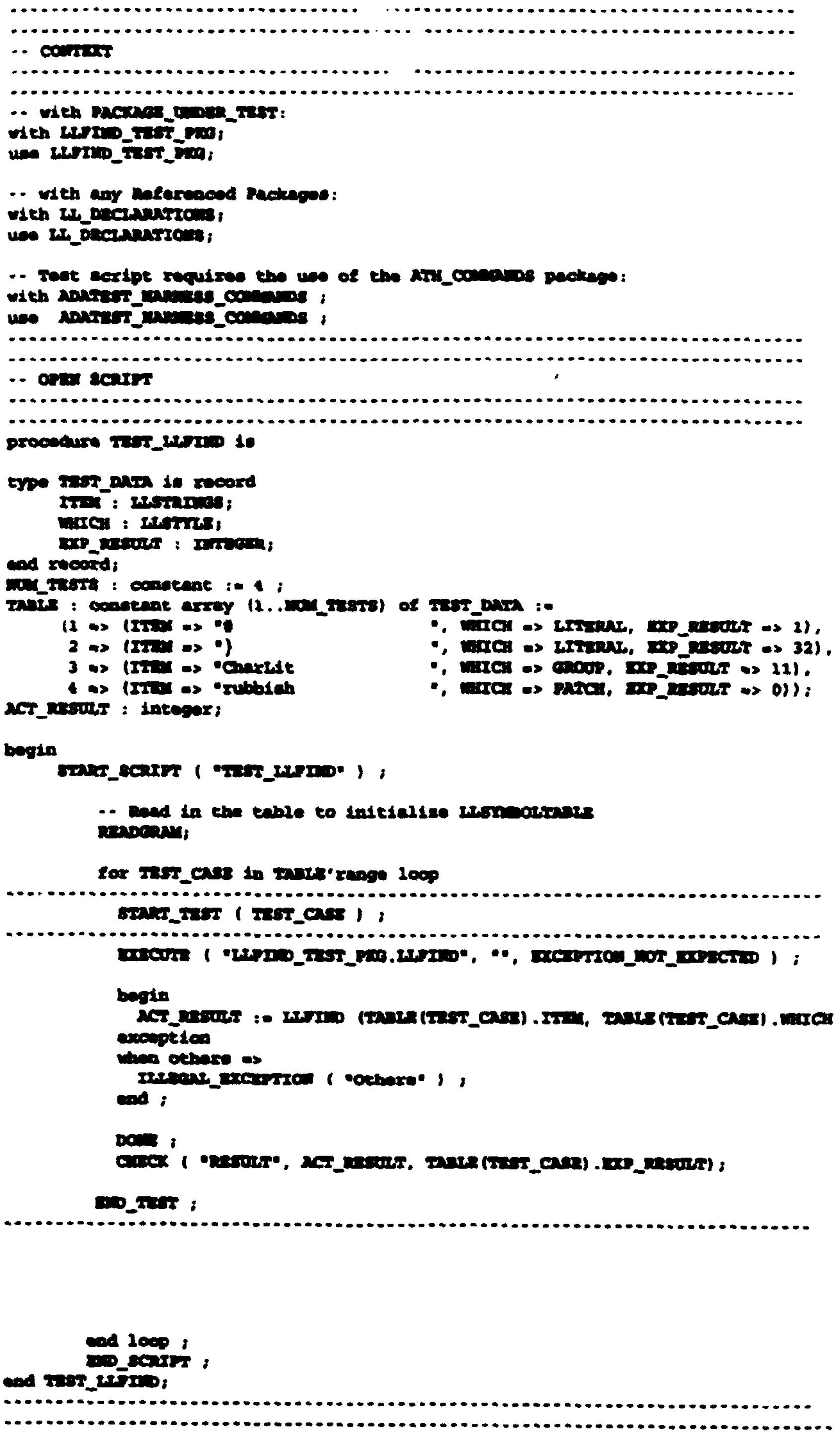
```
=========================================================================
        AdaTEST Harness        (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
-------------------------------------------------------------------------

        Test Results for       : TEST_LLFIND
        System Configured for  : SUN3_UNIX
        Test Run on  12 MAY 1993  at  13:25:55

=========================================================================

START_SCRIPT: TEST_LLFIND , AUTO ;

    RESET_TIMER ;

---------------------------------- TEST 1 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

            START_TIMER ;

            STOP_TIMER ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item       1
        CHECK_TIMER: ( SUN3_UNIX ) ;PASSED
        TIMER_VALUE = 0.020 seconds.

-------------------------------- END TEST 1 --------------------------------

---------------------------------- TEST 2 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item       32
-------------------------------- END TEST 2 --------------------------------

---------------------------------- TEST 3 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item       11
```

Figure 30-4. ATH Results of Testing LLFIND with Timing Analysis



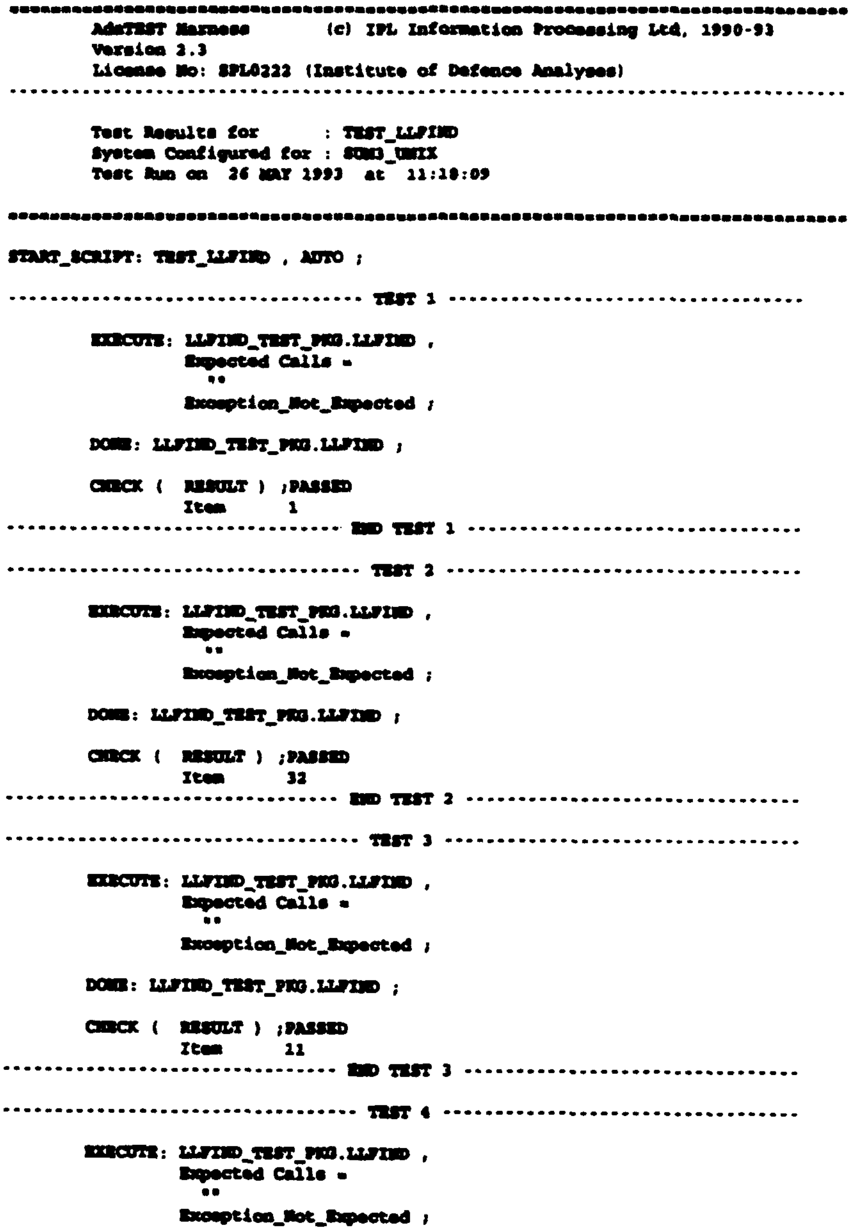
```
---------------------------------- END TEST 3 -------------------------------

---------------------------------- TEST 4 -----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item       0
---------------------------------- END TEST 4 -------------------------------

END_SCRIPT: TEST_LLFIND ;

================================================================================
        Test Results for TEST_LLFIND
        Test run completed at 13:25:56

        Script Errors              :  0
        Checks Passed              :  5
        Checks Failed              :  0
        Checks Ignored             :  0
        Paths with Stub Failures   :  0
--------------------------------------------------------------------------------
                    Overall Test PASSED
================================================================================
```

Figure 30-4 continued: Results of Testing LLFIND with Timing Analysis



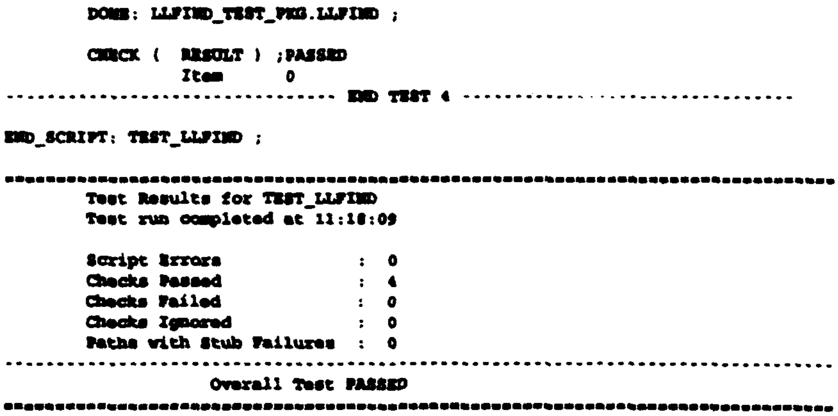
```
----------------------------------  ----------------------------------------
-------------------------------------------- -------------------------------
-- CONTEXT
-------------------------------------  ------------------------------------
----------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with LLFIND_TEST_PKG;
use LLFIND_TEST_PKG;

-- with any Referenced Packages:
with LL_DECLARATIONS;
use LL_DECLARATIONS;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- OPEN SCRIPT
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
procedure TEST_LLFIND is

type TEST_DATA is record
     ITEM : LLSTRINGS;
     WHICH : LLSTYLE;
     EXP_RESULT : INTEGER;
end record;
NUM_TESTS : constant := 4 ;
TABLE : constant array (1..NUM_TESTS) of TEST_DATA :=
     (1 => (ITEM => "$                    ", WHICH => LITERAL, EXP_RESULT => 1),
      2 => (ITEM => ")                    ", WHICH => LITERAL, EXP_RESULT => 32),
      3 => (ITEM => "CharLit              ", WHICH => GROUP, EXP_RESULT => 11),
      4 => (ITEM => "rubbish              ", WHICH => PATCH, EXP_RESULT => 0));
ACT_RESULT : integer;

begin
     START_SCRIPT ( "TEST_LLFIND" ) ;

         -- Read in the table to initialise LLSYMBOLTABLE
         READGRAM;

         for TEST_CASE in TABLE'range loop
-------------------------------------------------------------------------------
           START_TEST ( TEST_CASE ) ;
-------------------------------------------------------------------------------
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED ) ;

           begin
             ACT_RESULT := LLFIND (TABLE(TEST_CASE).ITEM, TABLE(TEST_CASE).WHICH
           exception
           when others =>
             ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;

           DONE ;
           CHECK ( "RESULT", ACT_RESULT, TABLE(TEST_CASE).EXP_RESULT);

         END_TEST ;
-----------------------------------------------------------------------------



        end loop ;
        END_SCRIPT ;
end TEST_LLFIND;
-----------------------------------------------------------------------------
-------------------------------------------------------------------------------
```

Figure 30-5. ATH Test Script for Table-Driven Testing of LLFIND



```
===================================================================================
        AdaTEST Harness         (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
-----------------------------------------------------------------------------------

        Test Results for       : TEST_LLFIND
        System Configured for  : SUN3_UNIX
        Test Run on  26 MAY 1993  at  11:18:09

===================================================================================

START_SCRIPT: TEST_LLFIND , AUTO ;

----------------------------------- TEST 1 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item      1
--------------------------------- END TEST 1 --------------------------------

----------------------------------- TEST 2 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item      32
--------------------------------- END TEST 2 --------------------------------

----------------------------------- TEST 3 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item      11
--------------------------------- END TEST 3 --------------------------------

----------------------------------- TEST 4 ----------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;
```

Figure 30-6. ATH Results of Table-Driven Testing of LLFIND



```
        DONE: LLFIND_TEST_PKG.LLFIND ;

        CHECK (  RESULT ) ;PASSED
                 Item      0
-------------------------------- END TEST 4 ---------------------------------

END_SCRIPT: TEST_LLFIND ;

==============================================================================
        Test Results for TEST_LLFIND
        Test run completed at 11:18:09

        Script Errors            :  0
        Checks Passed            :  4
        Checks Failed            :  0
        Checks Ignored           :  0
        Paths with Stub Failures :  0
------------------------------------------------------------------------------
                    Overall Test PASSED
==============================================================================
```

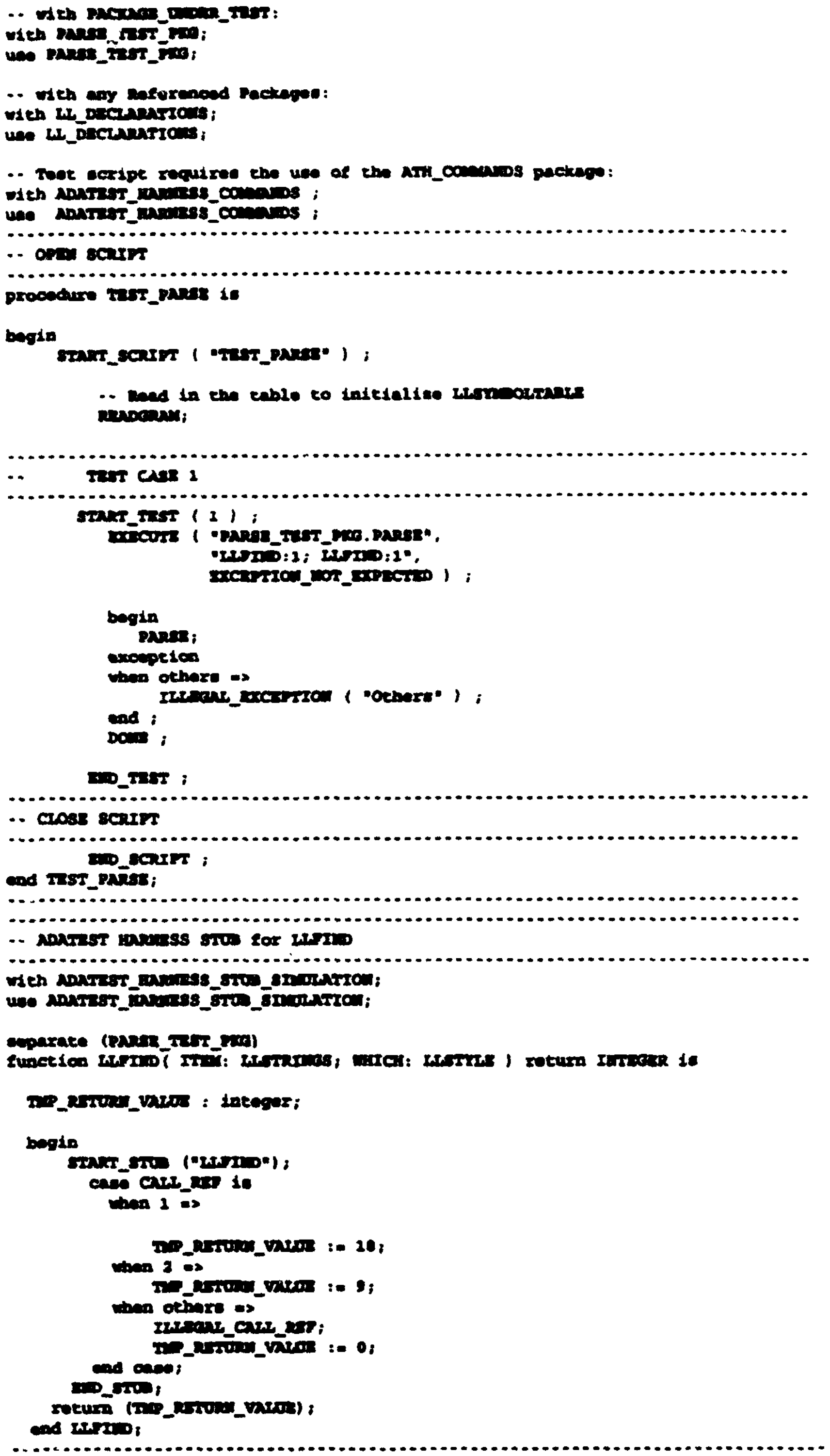
Figure 30-6 continued: Results of Table-Driven Testing of LLFIND

```
with  LL_DECLARATIONS, TEXT_IO;
use  LL_DECLARATIONS, TEXT_IO;

separate (PARSE_TEST_PKG)
procedure PARSE is

PARSING_ERROR: exception;   --  for fatal parsing errors

LLMAXSTACK: constant := 500;
   -- max number of sentential form elements in parse tree at one time

type  LLRIGHT  is     -- for grammar vocabulary symbols
   record
      CASEINDEX: INTEGER;     -- action code case index
      SYNCHINDEX: INTEGER;    -- synchronization table index
      WHICHCHILD: INTEGER;    -- position in production right hand side
      KIND: LLSTYLE;          -- type of vocabulary symbol
      TABLEINDEX: INTEGER;    -- symbol table or production start index
   end record;

type  LLSENTENTIAL  is     -- for sentential forms
   record
      LASTCHILD: BOOLEAN;        -- is this the rightmost child?
      TOP: INTEGER;              -- pointer to lastchild
      PARENT: INTEGER;           -- pointer to parent of this node
      ATTRIBUTE: LLATTRIBUTE;    -- derived attributes returned
      DATA: LLRIGHT;             -- vocabulary symbol information
   end record;

LLTOP: INTEGER;        -- top of stack pointer
LOCOFANY: INTEGER;     -- location of "any" in llsymboltable
LLLOCEOS: INTEGER;      -- location of end-of-input in symboltable
LLSENTPTR: INTEGER;     -- current sentential form element
LLSTACK: array ( 1 .. LLMAXSTACK ) of LLSENTENTIAL;
                        -- stack which represents the parse tree


begin
   LLSENTPTR := 1;
   LLTOP := 1;
   LLSTACK(LLSENTPTR).LASTCHILD := TRUE;
   LLSTACK(LLSENTPTR).PARENT := 0;
   LLSTACK(LLSENTPTR).DATA.SYNCHINDEX := 0;
   LLSTACK(LLSENTPTR).DATA.KIND := NONTERMINAL;
   -- find location of "any" in llsymboltable
   LOCOFANY := LLFIND( (1=>'a', 2=>'n', 3=>'y', others => ' '), GROUP );
   -- find location of endofsource ("@") in llsymboltable
   LLLOCEOS := LLFIND( (1=>'@', others => ' '), GROUP );
--                          ...
end  PARSE;
```

Figure 30-7. ATH Example Invocations of LLFIND

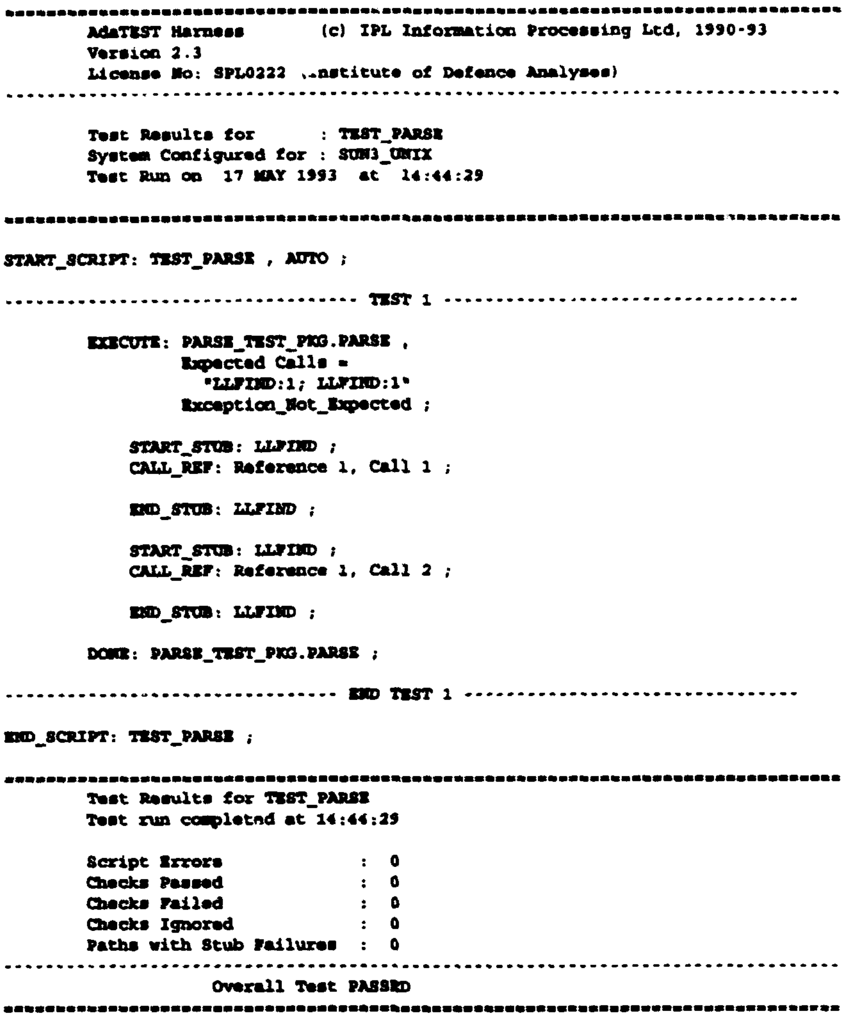
```
with  LL_DECLARATIONS, TEXT_IO;
package PARSE_TEST_PKG is
use  LL_DECLARATIONS, TEXT_IO;

type  LLSTYLE  is  (LITERAL, NONTERMINAL, GROUP, ACTION, PATCH);

type  LLSYMTABENTRY  is      -- for symbol table entries
   record
      KEY: LLSTRINGS;    -- literal string or group identifier
      KIND: LLSTYLE;     -- literal or group
   end record;

LLSYMBOLTABLE: array ( 1 .. LLTABLESIZE ) of LLSYMTABENTRY;
                          -- the symbol table for literal terms

procedure PARSE;
function LLFIND ( ITEM: LLSTRINGS; WHICH: LLSTYLE ) return INTEGER;
procedure READGRAM;

end PARSE_TRST_PKG;


package body PARSE_TEST_PKG is

    procedure PARSE is separate;
    function LLFIND ( ITEM: LLSTRINGS; WHICH: LLSTYLE ) return INTEGER is
                            separate;

    procedure  READGRAM  is      -- read grammar from disk
       CH: CHARACTER;
       LLGRAM: FILE_TYPE;      -- where grammar is stored
    begin   -- READGRAM
       OPEN( LLGRAM, IN_FILE, "TABLE" );
       -- read in symbol tables
       for  I  in  1 .. LLTABLESIZE  loop
          for  J  in  1 .. LLSTRINGLENGTH  loop
             GET( LLGRAM, LLSYMBOLTABLE(I).KEY(J) );
          end loop;
          GET( LLGRAM, CH );
          SKIP_LINE( LLGRAM );
          if  CH = 'g'  then
             LLSYMBOLTABLE(I).KIND := GROUP;
          else   -- assume ch = 1
             LLSYMBOLTABLE(I).KIND := LITERAL;
          end if;
       end loop;
       CLOSE( LLGRAM );
    end  READGRAM;

end PARSE_TEST_PKG;
```

Figure 30-8. AdaTEST Declaring PARSE and LLFIND as Separate



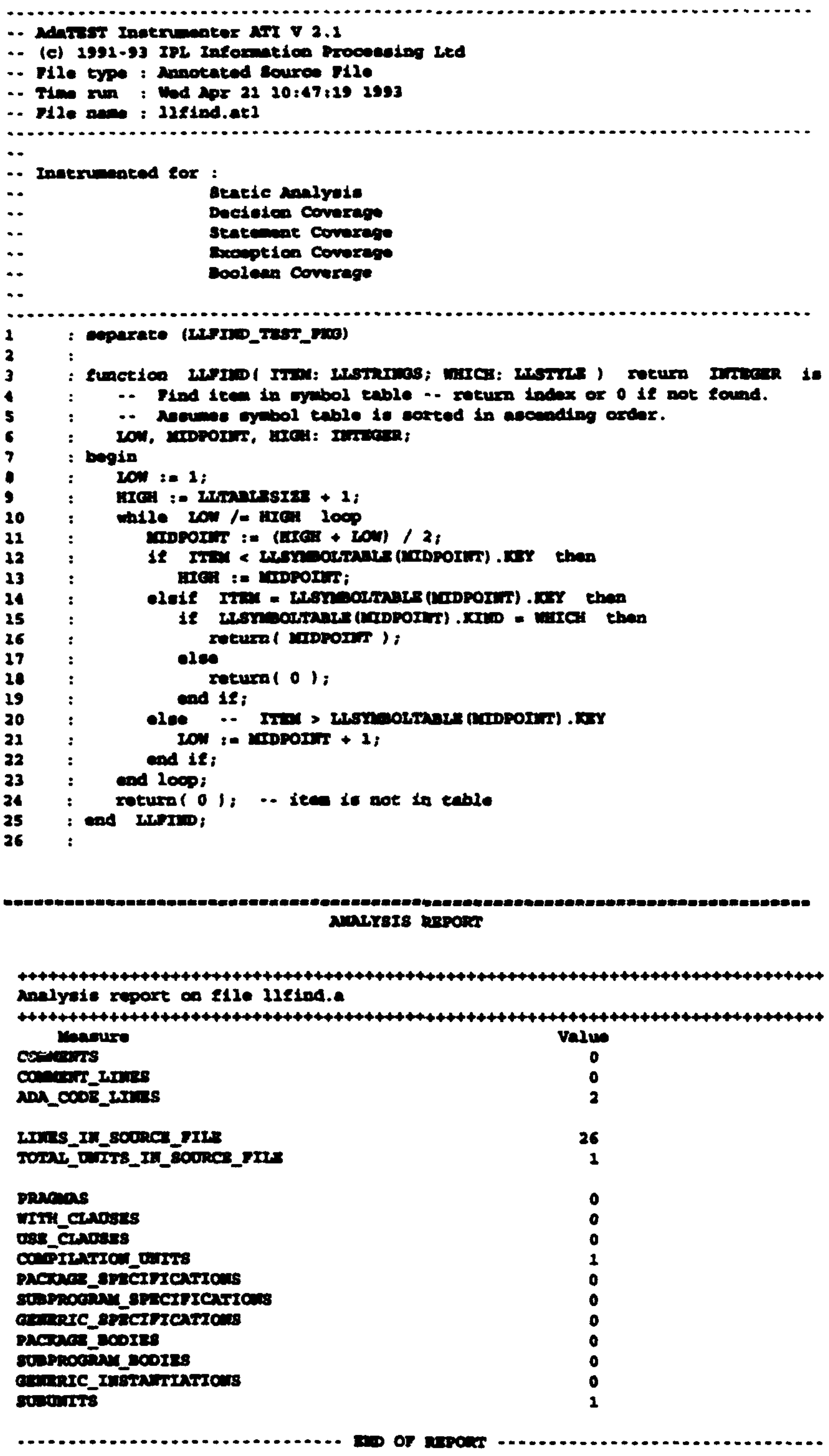
```
-- with PACKAGE_UNDER_TEST:
with PARSE_TEST_PKG;
use PARSE_TEST_PKG;

-- with any Referenced Packages:
with LL_DECLARATIONS;
use LL_DECLARATIONS;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
----------------------------------------------------------------------
-- OPEN SCRIPT
----------------------------------------------------------------------
procedure TEST_PARSE is

begin
     START_SCRIPT ( "TEST_PARSE" ) ;

         -- Read in the table to initialise LLSYMBOLTABLE
         READGRAM;

----------------------------------------------------------------------
--      TEST CASE 1
----------------------------------------------------------------------
       START_TEST ( 1 ) ;
          EXECUTE ( "PARSE_TEST_PKG.PARSE",
                    "LLFIND:1; LLFIND:1",
                    EXCEPTION_NOT_EXPECTED ) ;

          begin
             PARSE;
          exception
          when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
          end ;
          DONE ;

        END_TEST ;
----------------------------------------------------------------------
-- CLOSE SCRIPT
----------------------------------------------------------------------
        END_SCRIPT ;
end TEST_PARSE;
----------------------------------------------------------------------
----------------------------------------------------------------------
-- ADATEST HARNESS STUB for LLFIND
----------------------------------------------------------------------
with ADATEST_HARNESS_STUB_SIMULATION;
use ADATEST_HARNESS_STUB_SIMULATION;

separate (PARSE_TEST_PKG)
function LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE ) return INTEGER is

  TMP_RETURN_VALUE : integer;

  begin
      START_STUB ("LLFIND");
        case CALL_REF is
          when 1 =>

              TMP_RETURN_VALUE := 10;
          when 2 =>
              TMP_RETURN_VALUE := 9;
          when others =>
              ILLEGAL_CALL_REF;
              TMP_RETURN_VALUE := 0;
        end case;
      END_STUB;
    return (TMP_RETURN_VALUE);
  end LLFIND;
----------------------------------------------------------------------
```

Figure 30-9. ATH Test Script Showing Stub Simulation for LLFIND



```
========================================================================
       AdaTEST Harness        (c) IPL Information Processing Ltd, 1990-93
       Version 2.3
       License No: SPL0222 (Institute of Defence Analyses)
------------------------------------------------------------------------

       Test Results for       : TEST_PARSE
       System Configured for : SUN3_UNIX
       Test Run on  17 MAY 1993  at  14:44:29

========================================================================

START_SCRIPT: TEST_PARSE , AUTO ;

--------------------------------- TEST 1 ---------------------------------

       EXECUTE: PARSE_TEST_PKG.PARSE ,
                Expected Calls =
                  "LLFIND:1; LLFIND:1"
                Exception_Not_Expected ;

           START_STUB: LLFIND ;
           CALL_REF: Reference 1, Call 1 ;

           END_STUB: LLFIND ;

           START_STUB: LLFIND ;
           CALL_REF: Reference 1, Call 2 ;

           END_STUB: LLFIND ;

       DONE: PARSE_TEST_PKG.PARSE ;

------------------------------- END TEST 1 -------------------------------

END_SCRIPT: TEST_PARSE ;

========================================================================
       Test Results for TEST_PARSE
       Test run completnd at 14:44:29

       Script Errors            :  0
       Checks Passed            :  0
       Checks Failed            :  0
       Checks Ignored           :  0
       Paths with Stub Failures :  0
------------------------------------------------------------------------
                  Overall Test PASSED
========================================================================
```

Figure 30-10. ATH Results of Test with LLFIND Stub Invocation

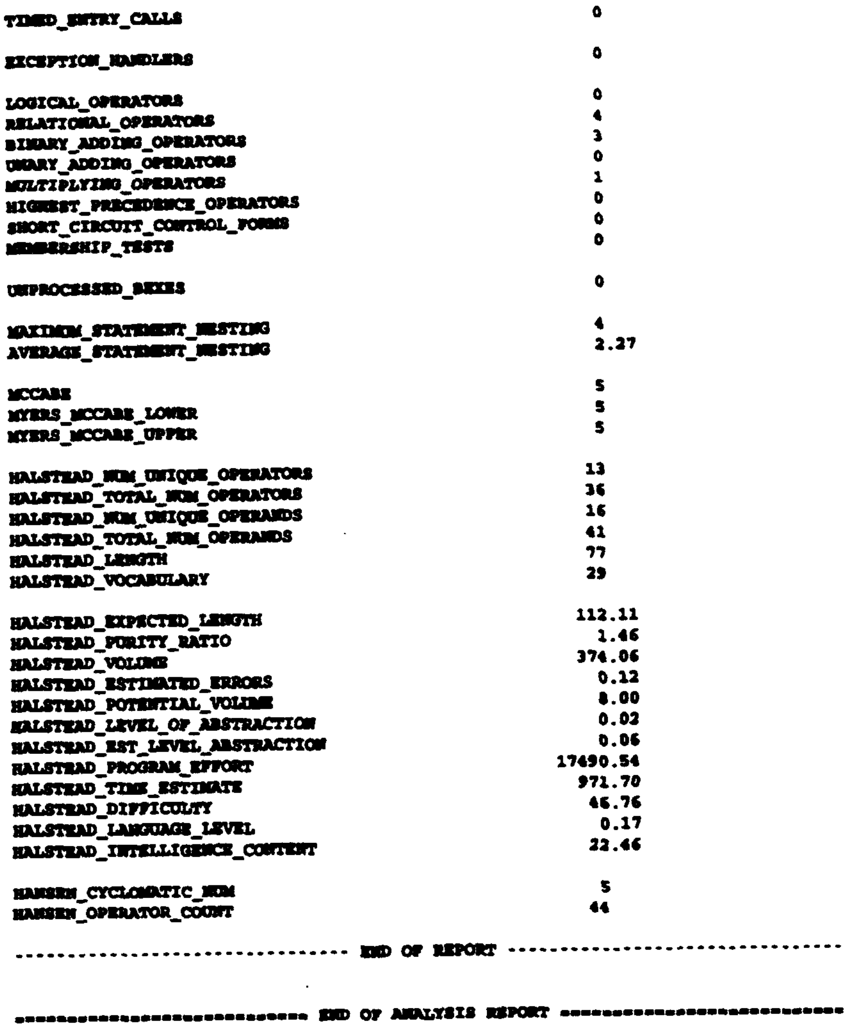


```
-------------------------------------------------------------------------------
-- AdaTEST Instrumenter ATI V 2.1
-- (c) 1991-93 IPL Information Processing Ltd
-- File type : Annotated Source File
-- Time run  : Wed Apr 21 10:47:19 1993
-- File name : llfind.atl
-------------------------------------------------------------------------------
--
-- Instrumented for :
--                     Static Analysis
--                     Decision Coverage
--                     Statement Coverage
--                     Exception Coverage
--                     Boolean Coverage
--
-------------------------------------------------------------------------------
1      : separate (LLFIND_TEST_PKG)
2      :
3      : function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
4      :    --  Find item in symbol table -- return index or 0 if not found.
5      :    --  Assumes symbol table is sorted in ascending order.
6      :    LOW, MIDPOINT, HIGH: INTEGER;
7      : begin
8      :    LOW := 1;
9      :    HIGH := LLTABLESIZE + 1;
10     :    while  LOW /= HIGH  loop
11     :       MIDPOINT := (HIGH + LOW) / 2;
12     :       if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
13     :          HIGH := MIDPOINT;
14     :       elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
15     :          if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
16     :             return( MIDPOINT );
17     :          else
18     :             return( 0 );
19     :          end if;
20     :       else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
21     :          LOW := MIDPOINT + 1;
22     :       end if;
23     :    end loop;
24     :    return( 0 );  -- item is not in table
25     : end  LLFIND;
26     :

===============================================================================
                               ANALYSIS REPORT

 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 Analysis report on file llfind.a
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
     Measure                                                 Value
 COMMENTS                                                      0
 COMMENT_LINES                                                 0
 ADA_CODE_LINES                                                2

 LINES_IN_SOURCE_FILE                                         26
 TOTAL_UNITS_IN_SOURCE_FILE                                    1

 PRAGMAS                                                       0
 WITH_CLAUSES                                                  0
 USE_CLAUSES                                                   0
 COMPILATION_UNITS                                             1
 PACKAGE_SPECIFICATIONS                                        0
 SUBPROGRAM_SPECIFICATIONS                                     0
 GENERIC_SPECIFICATIONS                                        0
 PACKAGE_BODIES                                                0
 SUBPROGRAM_BODIES                                             0
 GENERIC_INSTANTIATIONS                                        0
 SUBUNITS                                                      1

 ------------------------------ END OF REPORT ------------------------------
```

Figure 30-11. ATI Static Analysis of LLFIND

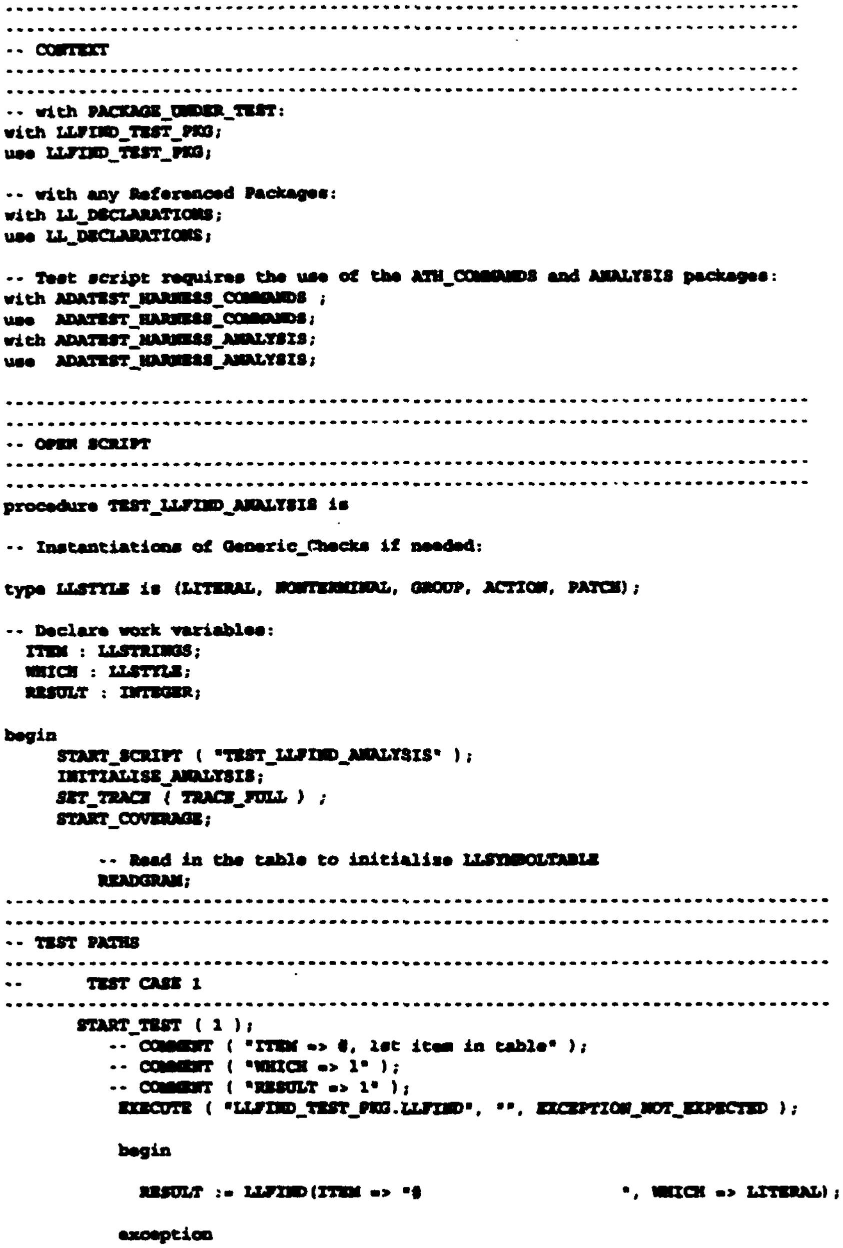
```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on function LLFIND_TEST_PKG.LLFIND
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                                Value
TOTAL_LINES                                                 22
COMMENTS                                                     4
COMMENT_LINES                                                2
ADA_CODE_LINES                                              20

LINES_IN_SOURCE_FILE                                        26
TOTAL_UNITS_IN_SOURCE_FILE                                   1

PRAGMAS                                                      0
WITH_CLAUSES                                                 0
USE_CLAUSES                                                  0
INNER_USE_CLAUSES                                            0

PARAMETERS                                                   2
DECLARATIVE_STATEMENTS                                       1
EXECUTABLE_STATEMENTS                                       11

GENERIC_DECLARATIONS                                         0
GENERIC_INSTANTIATIONS                                       0
UNCHECKED_DEALLOCATIONS                                      0
UNCHECKED_CONVERSIONS                                        0
BODY_STUBS                                                   0
DERIVED_TYPE_DEFINITIONS                                     0
ALLOCATORS                                                   0

ASSIGNMENT_STATEMENTS                                        5
PROCEDURE_ENTRY_CALL_STMTS                                   0
EXIT_STATEMENTS                                              0
EXIT_WHEN_PARTS                                              0
RETURN_STATEMENTS                                            3
GOTO_STATEMENTS                                              0
DELAY_STATEMENTS                                             0
ABORT_STATEMENTS                                             0
RAISE_STATEMENTS                                             0
CODE_STATEMENTS                                              0

IF_STATEMENTS                                                2
ELSIF_PARTS                                                  1
ELSE_PARTS                                                   2
CASE_STATEMENTS                                              0
CASE_ALTERNATIVES                                            0

LOOP_STATEMENTS                                              1
FOR_LOOPS                                                    0
WHILE_LOOPS                                                  1

BLOCK_STATEMENTS                                             0
BLOCK_DECLARATIVE_PARTS                                      0

ACCEPT_STATEMENTS                                            0
SELECT_STATEMENTS                                            0
SELECTIVE_WAITS                                              0
SELECTIVE_WAIT_ALTERNATIVES                                  0
TERMINATES                                                   0
CONDITIONAL_ENTRY_CALLS                                      0
```

Figure 30-11 continued: ATI Static Analysis of LLFIND



```
TIMED_ENTRY_CALLS                                       0

EXCEPTION_HANDLERS                                      0

LOGICAL_OPERATORS                                       0
RELATIONAL_OPERATORS                                    4
BINARY_ADDING_OPERATORS                                 3
UNARY_ADDING_OPERATORS                                  0
MULTIPLYING_OPERATORS                                   1
HIGHEST_PRECEDENCE_OPERATORS                            0
SHORT_CIRCUIT_CONTROL_FORMS                             0
MEMBERSHIP_TESTS                                        0

UNPROCESSED_BEXES                                       0

MAXIMUM_STATEMENT_NESTING                               4
AVERAGE_STATEMENT_NESTING                               2.27

MCCABE                                                  5
MYERS_MCCABE_LOWER                                      5
MYERS_MCCABE_UPPER                                      5

HALSTEAD_NUM_UNIQUE_OPERATORS                          13
HALSTEAD_TOTAL_NUM_OPERATORS                           36
HALSTEAD_NUM_UNIQUE_OPERANDS                           16
HALSTEAD_TOTAL_NUM_OPERANDS                            41
HALSTEAD_LENGTH                                        77
HALSTEAD_VOCABULARY                                    29

HALSTEAD_EXPECTED_LENGTH                              112.11
HALSTEAD_PURITY_RATIO                                   1.46
HALSTEAD_VOLUME                                       374.06
HALSTEAD_ESTIMATED_ERRORS                               0.12
HALSTEAD_POTENTIAL_VOLUME                               8.00
HALSTEAD_LEVEL_OF_ABSTRACTION                           0.02
HALSTEAD_EST_LEVEL_ABSTRACTION                          0.06
HALSTEAD_PROGRAM_EFFORT                             17490.54
HALSTEAD_TIME_ESTIMATE                                971.70
HALSTEAD_DIFFICULTY                                    46.76
HALSTEAD_LANGUAGE_LEVEL                                 0.17
HALSTEAD_INTELLIGENCE_CONTENT                          22.46

HANSEN_CYCLOMATIC_NUM                                   5
HANSEN_OPERATOR_COUNT                                  44

--------------------------------- END OF REPORT --------------------------------

============================ END OF ANALYSIS REPORT ==========================
```

Figure 30-11 continued: ATI Static Analysis of LLFIND



```
---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- CONTEXT
---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with LLFIND_TEST_PKG;
use LLFIND_TEST_PKG;

-- with any Referenced Packages:
with LL_DECLARATIONS;
use LL_DECLARATIONS;

-- Test script requires the use of the ATH_COMMANDS and ANALYSIS packages:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS;
with ADATEST_HARNESS_ANALYSIS;
use  ADATEST_HARNESS_ANALYSIS;

----------------------------------------------------------------------------
----------------------------------------------------------------------------
-- OPEN SCRIPT
----------------------------------------------------------------------------
----------------------------------------------------------------------------
procedure TEST_LLFIND_ANALYSIS is

-- Instantiations of Generic_Checks if needed:

type LLSTYLE is (LITERAL, NONTERMINAL, GROUP, ACTION, PATCH);

-- Declare work variables:
  ITEM : LLSTRINGS;
  WHICH : LLSTYLE;
  RESULT : INTEGER;

begin
     START_SCRIPT ( "TEST_LLFIND_ANALYSIS" );
     INITIALISE_ANALYSIS;
     SET_TRACE ( TRACE_FULL ) ;
     START_COVERAGE;

         -- Read in the table to initialise LLSYMBOLTABLE
         READGRAM;
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- TEST PATHS
-----------------------------------------------------------------------------
--      TEST CASE 1
-----------------------------------------------------------------------------
       START_TEST ( 1 );
          -- COMMENT ( "ITEM => $, 1st item in table" );
          -- COMMENT ( "WHICH => 1" );
          -- COMMENT ( "RESULT => 1" );
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED );

           begin

             RESULT := LLFIND(ITEM => "$                    ", WHICH => LITERAL);

           exception
```

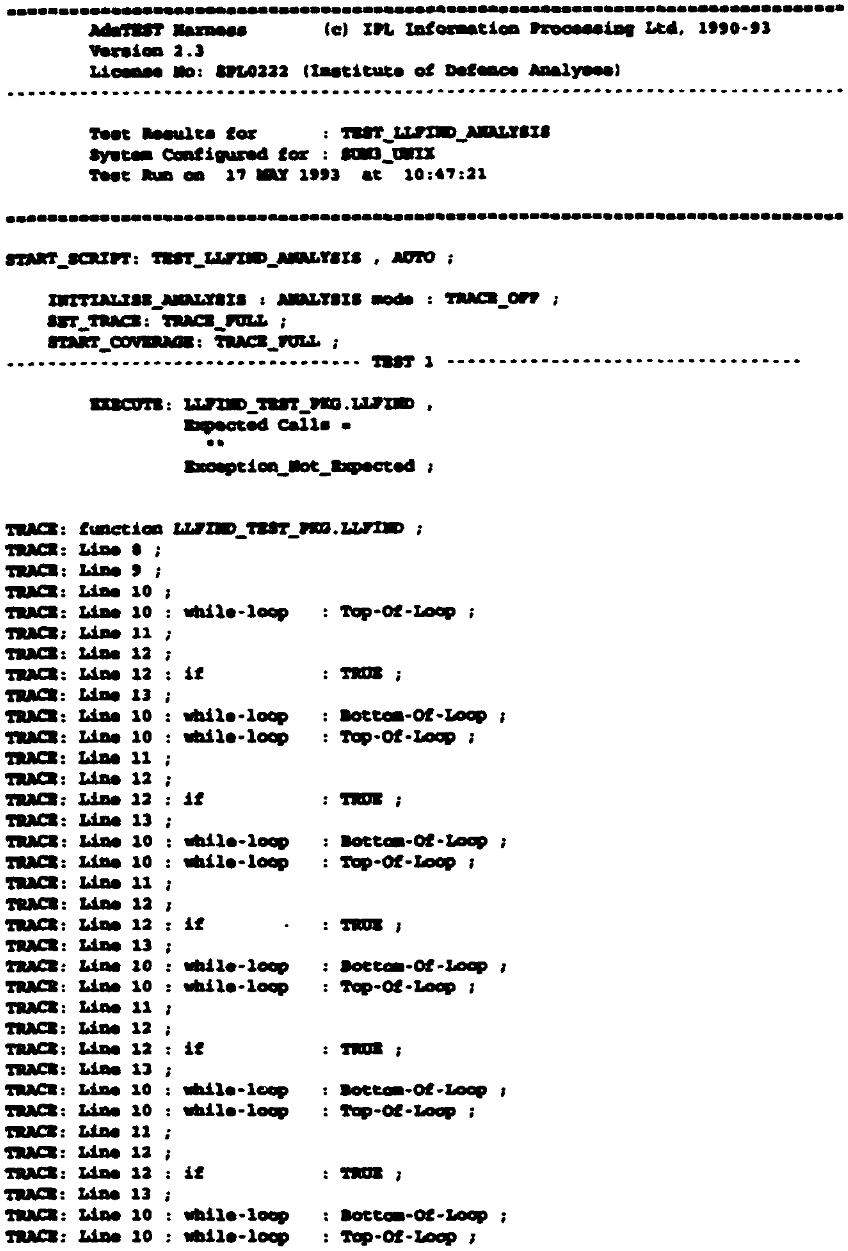
Figure 30-12. ATH Test Script for Coverage Analysis and Full Tracing of LLFIND

```
           when others =>
               ILLEGAL_EXCEPTION ( "Others" );
           end;
           DONE;
           CHECK ( "RESULT", RESULT, 1);

         END_TEST;

-- Switch off tracing for the remainder of the test
         STOP_COVERAGE;
         SET_TRACE ( TRACE_OFF );
         START_COVERAGE;
---------------------------------------------------------------------------------
--      TEST CASE 2
---------------------------------------------------------------------------------
       START_TEST ( 2 );
          -- COMMENT ( "ITEM => }, last item in table" );
          -- COMMENT ( "WHICH => 1" );
          -- COMMENT ( "RESULT => 32" );
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED );

           begin
             RESULT := LLFIND(ITEM => "}                    ", WHICH => LITERAL);
           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" );
           end;
           DONE;
           CHECK ( "RESULT", RESULT, 32);

         END_TEST;
---------------------------------------------------------------------------------
--      TEST CASE 3
---------------------------------------------------------------------------------
       START_TEST ( 3 );
          -- COMMENT ( "ITEM => CharLit, some random entry" );
          -- COMMENT ( "WHICH => GROUP" );
          -- COMMENT ( "RESULT => 11" );
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED );

           begin
             RESULT := LLFIND(ITEM => "CharLit              ", WHICH => GROUP);
           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" );
           end;
           DONE;
           CHECK ( "RESULT", RESULT, 11);

         END_TEST;
---------------------------------------------------------------------------------
--      TEST CASE 4
-------------------- ------------------------------------------------------------
       START_TEST ( 4 );
          -- COMMENT ( "ITEM => rubbish, item not present in table" );
          -- COMMENT ( "WHICH => LITERAL, valid LLSTYLE" );
          -- COMMENT ( "RESULT => 0" );
           EXECUTE ( "LLFIND_TEST_PKG.LLFIND", "", EXCEPTION_NOT_EXPECTED );

           begin
```

Figure 30-12 continued: ATH Test Script for Coverage Analysis and Full Tracing of LL-FIND

```
             RESULT := LLFIND(ITEM => "rubbish            ", WHICH => PATCH);
           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" );
           end;
           DONE;
           CHECK ( "RESULT", RESULT, 0);

         END_TEST;
----------------------------------------------------------------------------
    STOP_COVERAGE;

    -- Coverage analysis checks
    CHECK_ANALYSIS ("LLFIND_TEST_PKG.LLFIND",
                    "DECISION_COVERAGE",
                    100.00, 100.01);

    -- Static analysis checks
    CHECK_ANALYSIS ("LLFIND_TEST_PKG.LLFIND",
                    "USE_CLAUSES",
                    0.00, 0.01);

    CHECK_ANALYSIS ("LLFIND_TEST_PKG.LLFIND",
                    "MCCABE",
                    0.00, 10.0);

    -- Coverage analysis reports
    REPORT_UNIT ("LLFIND_TEST_PKG.LLFIND",
                 "STATEMENT_COVERAGE;" &
                 "STATEMENT_STATISTICS;" &
                 "DECISION_COVERAGE;" &
                 "DECISION_STATISTICS;" &
                 "BOOLEAN_OPERATOR_COVERAGE_3WAY;" &
                 "BOOLEAN_OPERATOR_STATISTICS_3WAY");

-----------------------------------------------------------------------------
-- CLOSE SCRIPT
----------------------------------------------------------------------------
----------------------------------------------------------------------------
      END_SCRIPT;
end TEST_LLFIND_ANALYSIS;
----------------------------------------------------------------------------
------------------------------------------------------------------------------
```

Figure 30-12 continued: ATH Test Script for Coverage Analysis and Full Tracing of LL-FIND

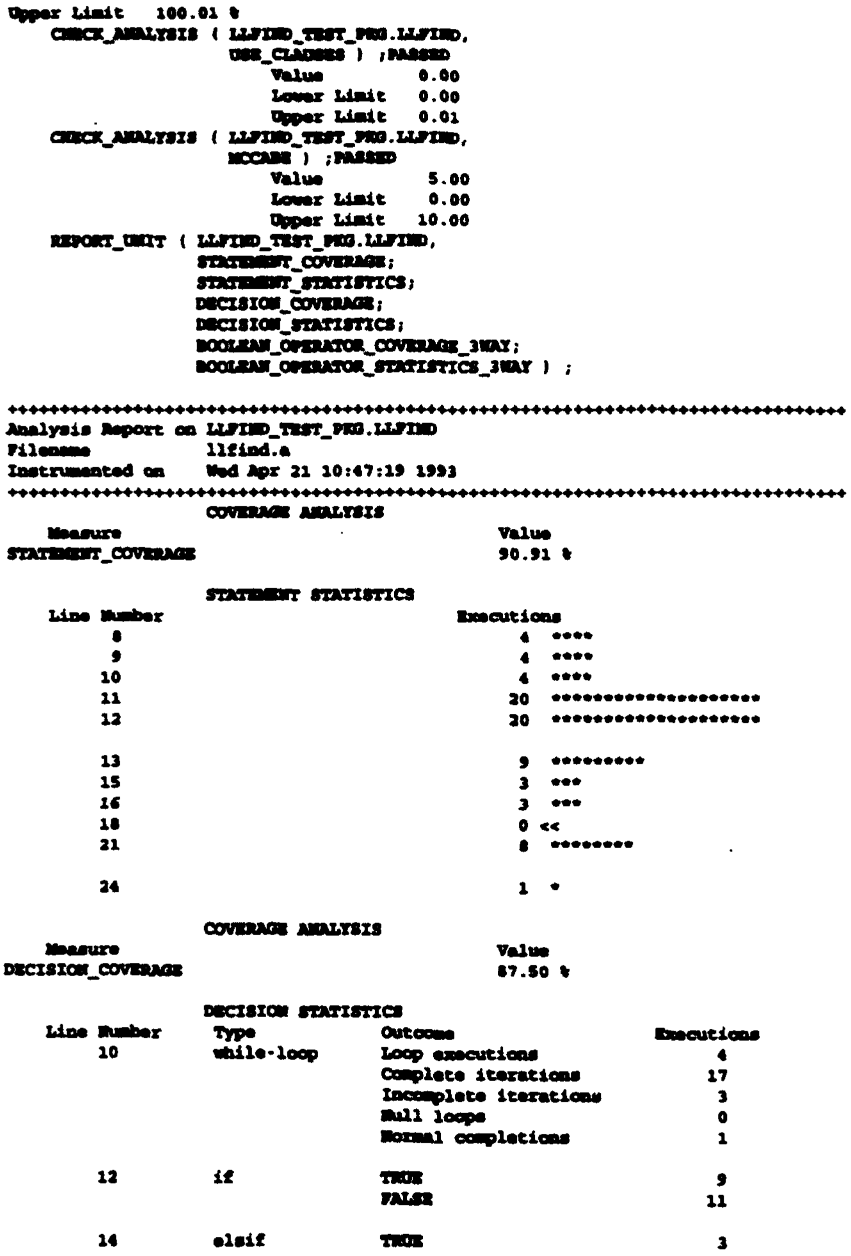


```
================================================================================
        AdaTEST Harness        (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
--------------------------------------------------------------------------------

        Test Results for       : TEST_LLFIND_ANALYSIS
        System Configured for  : SUN3_UNIX
        Test Run on  17 MAY 1993  at  10:47:21

================================================================================

START_SCRIPT: TEST_LLFIND_ANALYSIS , AUTO ;

    INITIALISE_ANALYSIS : ANALYSIS mode : TRACE_OFF ;
    SET_TRACE: TRACE_FULL ;
    START_COVERAGE: TRACE_FULL ;
--------------------------------- TEST 1 ---------------------------------

        EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;


TRACE: function LLFIND_TEST_PKG.LLFIND ;
TRACE: Line 8 ;
TRACE: Line 9 ;
TRACE: Line 10 ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if            : TRUE ;
TRACE: Line 13 ;
TRACE: Line 10 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if            : TRUE ;
TRACE: Line 13 ;
TRACE: Line 10 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if          . : TRUE ;
TRACE: Line 13 ;
TRACE: Line 10 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if            : TRUE ;
TRACE: Line 13 ;
TRACE: Line 10 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if            : TRUE ;
TRACE: Line 13 ;
TRACE: Line 10 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 10 : while-loop    : Top-Of-Loop ;
```

Figure 30-13. ATH Results of Coverage Analysis and Full Tracing of LLFIND

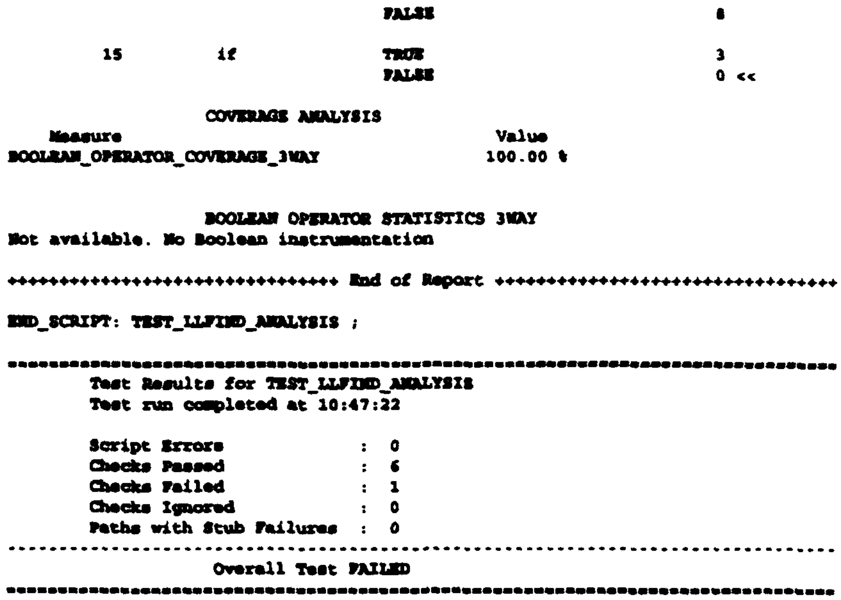
```
TRACE: Line 11 ;
TRACE: Line 12 ;
TRACE: Line 12 : if          : FALSE ;
TRACE: Line 14 : elsif       : TRUE ;
TRACE: Line 15 ;
TRACE: Line 15 : if          : TRUE ;
TRACE: Line 16 ;
       DONE: LLFIND_TEST_PKG.LLFIND ;

       CHECK (  RESULT ) ;PASSED
                Item      1
------------------------------- END TEST 1 -------------------------------

    STOP_COVERAGE ;
    SET_TRACE: TRACE_OFF ;
    START_COVERAGE: TRACE_OFF ;
--------------------------------- TEST 2 ---------------------------------

       EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                Expected Calls =
                  **
                Exception_Not_Expected ;

       DONE: LLFIND_TEST_PKG.LLFIND ;

       CHECK (  RESULT ) ;PASSED
                Item      32
------------------------------- END TEST 2 -------------------------------

--------------------------------- TEST 3 ---------------------------------

       EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                Expected Calls =
                  **
                Exception_Not_Expected ;

       DONE: LLFIND_TEST_PKG.LLFIND ;

       CHECK (  RESULT ) ;PASSED
                Item      11
------------------------------- END TEST 3 -------------------------------

--------------------------------- TEST 4 ---------------------------------

       EXECUTE: LLFIND_TEST_PKG.LLFIND ,
                Expected Calls =
                  **
                Exception_Not_Expected ;

       DONE: LLFIND_TEST_PKG.LLFIND ;

       CHECK (  RESULT ) ;PASSED
                Item      0
------------------------------- END TEST 4 -------------------------------

    STOP_COVERAGE ;
    CHECK_ANALYSIS ( LLFIND_TEST_PKG.LLFIND,
                     DECISION_COVERAGE ) ;>>FAILED
Value          87.50 %
Lower Limit   100.00 %
```

Figure 30-13 continued: ATH Results of Coverage Analysis and Full Tracing of LLFIND



```
Upper Limit   100.01 %
     CHECK_ANALYSIS ( LLFIND_TEST_PKG.LLFIND,
                      USE_CLAUSES ) ;PASSED
                           Value         0.00
                           Lower Limit   0.00
                           Upper Limit   0.01
     CHECK_ANALYSIS ( LLFIND_TEST_PKG.LLFIND,
                      MCCABE ) ;PASSED
                           Value          5.00
                           Lower Limit    0.00
                           Upper Limit   10.00
     REPORT_UNIT ( LLFIND_TEST_PKG.LLFIND,
                   STATEMENT_COVERAGE;
                   STATEMENT_STATISTICS;
                   DECISION_COVERAGE;
                   DECISION_STATISTICS;
                   BOOLEAN_OPERATOR_COVERAGE_3WAY;
                   BOOLEAN_OPERATOR_STATISTICS_3WAY ) ;

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report on LLFIND_TEST_PKG.LLFIND
Filename           llfind.a
Instrumented on    Wed Apr 21 10:47:19 1993
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                   COVERAGE ANALYSIS
    Measure                                        Value
STATEMENT_COVERAGE                                 90.91 %

                   STATEMENT STATISTICS
    Line Number                                Executions
         8                                         4 ****
         9                                         4 ****
        10                                         4 ****
        11                                        20 ********************
        12                                        20 ********************

        13                                         9 *********
        15                                         3 ***
        16                                         3 ***
        18                                         0 <<
        21                                         8 ********

        24                                         1 *

                   COVERAGE ANALYSIS
    Measure                                        Value
DECISION_COVERAGE                                  87.50 %

                   DECISION STATISTICS
    Line Number    Type          Outcome                     Executions
        10         while-loop    Loop executions                  4
                                 Complete iterations             17
                                 Incomplete iterations            3
                                 Null loops                       0
                                 Normal completions               1

        12         if            TRUE                             9
                                 FALSE                           11

        14         elsif         TRUE                             3
```

Figure 30-13 continued: ATH Results of Coverage Analysis and Full Tracing of LLFIND



```
                                          FALSE                        6

          15          if                  TRUE                         3
                                          FALSE                        0 <<

                    COVERAGE ANALYSIS
     Measure                                   Value
BOOLEAN_OPERATOR_COVERAGE_3WAY                100.00 %


                    BOOLEAN OPERATOR STATISTICS 3WAY
Not available. No Boolean instrumentation

++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++

END_SCRIPT: TEST_LLFIND_ANALYSIS ;

==============================================================================
        Test Results for TEST_LLFIND_ANALYSIS
        Test run completed at 10:47:22

        Script Errors            :  0
        Checks Passed            :  6
        Checks Failed            :  1
        Checks Ignored           :  0
        Paths with Stub Failures :  0
------------------------------------------------------------------------------
                  Overall Test FAILED
==============================================================================
```

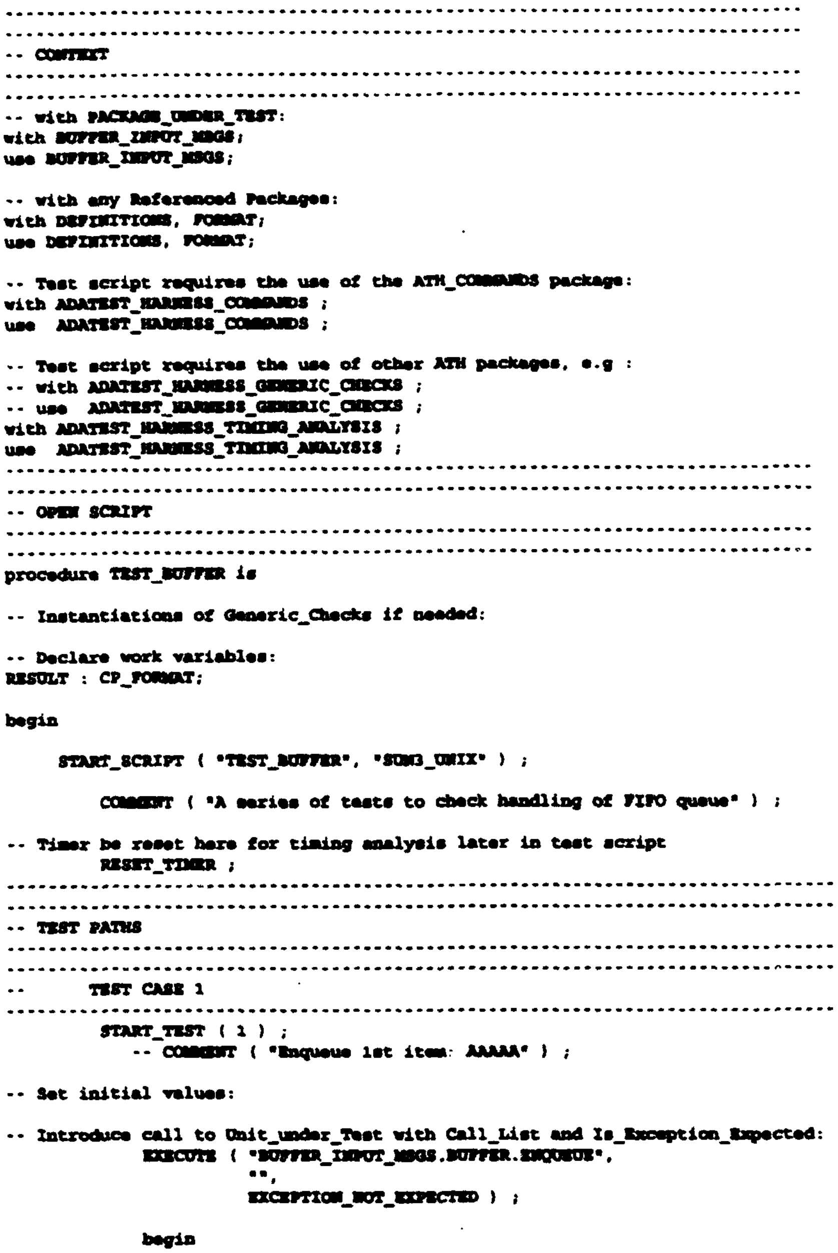
Figure 30-13 continued: ATH Results of Coverage Analysis and Full Tracing of LLFIND

```
-- buffer_input_msgs.a - non-generic code to work around VADS problem

  with Definitions; use Definitions;
  package BUFFER_INPUT_MSGS is

    SIZE : NATURAL:=MAX_CP_MSGS;

    procedure ENQUEUE (I:in  CP_FORMAT);
    procedure DEQUEUE (I:out CP_FORMAT);
    pragma INLINE (ENQUEUE,DEQUEUE);

  end BUFFER_INPUT_MSGS;

  with Text_Io;
  with Calendar;
  with Format;
  package body BUFFER_INPUT_MSGS is

    task BUFFER is
      entry ENQUEUE (I:in  CP_FORMAT);
      entry DEQUEUE (I:out CP_FORMAT);
    end BUFFER;

    task body BUFFER is

      subtype INDEX_TYPE is POSITIVE range 1..SIZE;
      subtype COUNT_TYPE is NATURAL  range 0..SIZE;
      BUF    : array (INDEX_TYPE) of CP_FORMAT;
      INSERT : INDEX_TYPE:=1;
      REMOVE : INDEX_TYPE:=1;
      COUNT  : COUNT_TYPE:=0;
      TEMP   : CP_FORMAT;

    begin
      while not Finished loop
         select
        accept ENQUEUE (I:in CP_FORMAT) do
          if Definitions.Debug then
            Text_Io.Put_Line (Format.Time_Image (Calendar.Clock) & "  " &
              "Buffer_Input_Msgs about to enqueue: " & I);
          end if;
          if COUNT<SIZE then  -- check if request ignored
            BUF(INSERT):=I;
            INSERT:=(INSERT mod BUF'LAST)+1;
            COUNT:=COUNT+1;
              end if;
        end ENQUEUE;
         or
        when COUNT>0 =>
          accept DEQUEUE(I:out CP_FORMAT) do
            TEMP:=BUF(REMOVE);
            I := TEMP;
            if Definitions.Debug then
              Text_Io.Put_Line (Format.Time_Image (Calendar.Clock) & "  " &
              "Buffer_Input_Msgs dequeued: " & TEMP);
            end if;
          end DEQUEUE;
          REMOVE:=(REMOVE mod BUF'LAST)+1;
          COUNT:=COUNT-1;
         or
```

Figure 30-14. ATHListing of BUFFER Package

```
        delay 10.0;
      end select;
    end loop;
  end BUFFER;

  procedure ENQUEUE (I:in CP_FORMAT) is
  begin
    select
      BUFFER.ENQUEUE(I);
    else
      null; -- Request ignored
    end select;
  end ENQUEUE;

  procedure DEQUEUE (I:out CP_FORMAT) is
  begin
    BUFFER.DEQUEUE(I);
  end DEQUEUE;

end BUFFER_INPUT_MSGS;
```

Figure 30-14 continued:ATH Listing of BUFFER Package



```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- CONTEXT
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with BUFFER_INPUT_MSGS;
use BUFFER_INPUT_MSGS;

-- with any Referenced Packages:
with DEFINITIONS, FORMAT;
use DEFINITIONS, FORMAT;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;

-- Test script requires the use of other ATH packages, e.g :
-- with ADATEST_HARNESS_GENERIC_CHECKS ;
-- use  ADATEST_HARNESS_GENERIC_CHECKS ;
with ADATEST_HARNESS_TIMING_ANALYSIS ;
use  ADATEST_HARNESS_TIMING_ANALYSIS ;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- OPEN SCRIPT
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
procedure TEST_BUFFER is

-- Instantiations of Generic_Checks if needed:

-- Declare work variables:
RESULT : CP_FORMAT;

begin

     START_SCRIPT ( "TEST_BUFFER", "SUN3_UNIX" ) ;

         COMMENT ( "A series of tests to check handling of FIFO queue" ) ;

-- Timer be reset here for timing analysis later in test script
         RESET_TIMER ;
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-- TEST PATHS
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
--      TEST CASE 1
----------------------------------------------------------------------------------
         START_TEST ( 1 ) ;
            -- COMMENT ( "Enqueue 1st item: AAAAA" ) ;

-- Set initial values:

-- Introduce call to Unit_under_Test with Call_List and Is_Exception_Expected:
             EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                       "",
                       EXCEPTION_NOT_EXPECTED ) ;

             begin
```

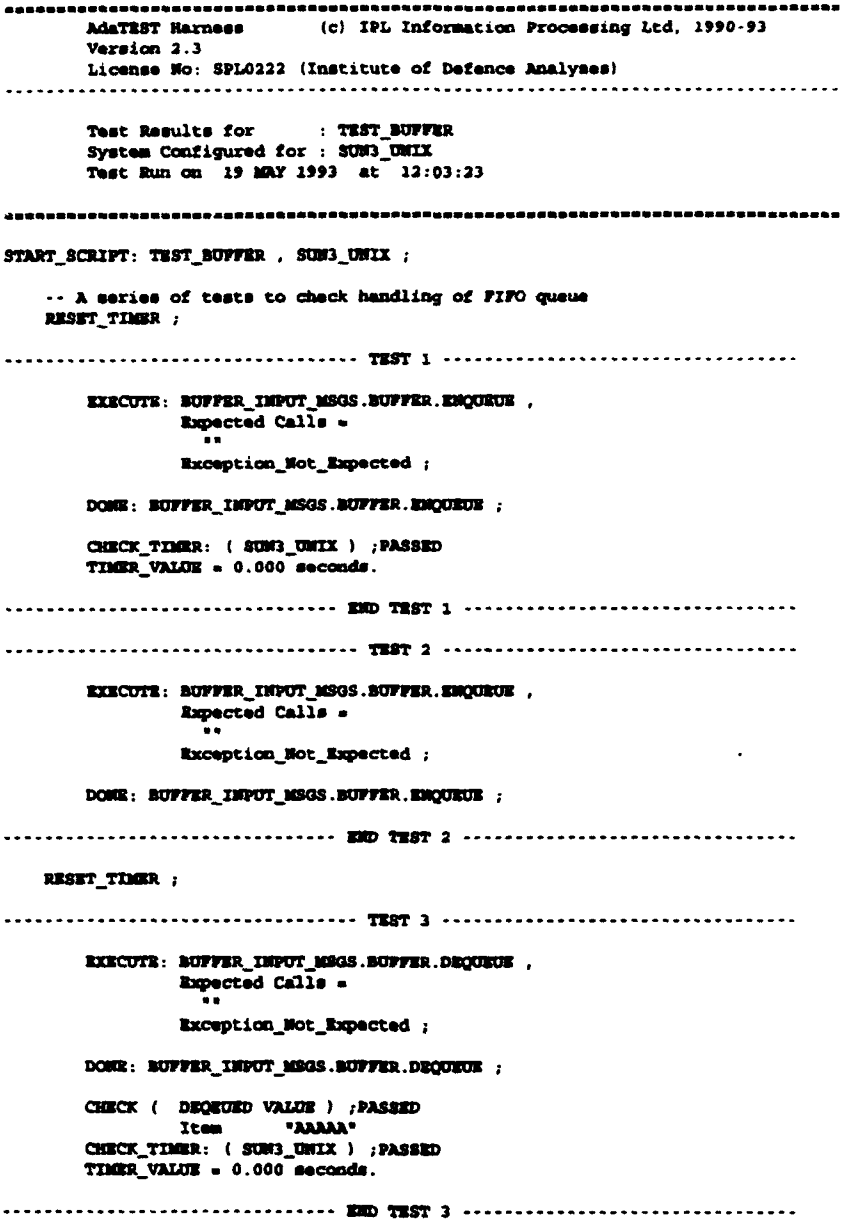
Figure 30-15. ATH Test Script for Testing Task BUFFER with Timing Analysis

```
-- Make call to Unit_under_Test with appropriate value(s) for In parameters
-- and work variables for Out parameters and returned values:
              BUFFER_INPUT_MSGS.ENQUEUE (I => "AAAAA");
              delay 1.0;  -- Allow processing to be performed

-- Exception handler(s):
            exception

        -- If test case requires, provide a Correct_Exception handler, e.g.
        -- when PACKAGE_UNDER_TEST.EXPECTED_EXCEPTION =>
        --      CORRECT_EXCEPTION ("EXPECTED_EXCEPTION");

        -- Always provide an ILLEGAL_EXCEPTION handler for at least 'others':
            when others =>
                ILLEGAL_EXCEPTION ( "Others" ) ;
            end ;

            DONE ;

-- Check the results of executing the Unit_under_test:
--      Out parameters/returned value
--      Global data
           CHECK_TIMER ("SUN3_UNIX", 0.0000, 0.0005);

        END_TEST ;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
--      TEST CASE 2
----------------------------------------------------------------------------
        START_TEST ( 2 ) ;
           -- COMMENT ( "Enqueue 2nd item: BBBBB" ) ;

            EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                      "",
                      EXCEPTION_NOT_EXPECTED ) ;

            begin

              BUFFER_INPUT_MSGS.ENQUEUE (I => "BBBBB");
              delay 1.0;  -- Allow processing to be performed

            exception
            when others =>
                ILLEGAL_EXCEPTION ( "Others" ) ;
            end ;

            DONE ;

        END_TEST ;
----------------------------------------------------------------------------
--      TEST CASE 3
----------------------------------------------------------------------------
        RESET_TIMER;
        START_TEST ( 3 ) ;
           -- COMMENT ( "Dequeue 1st queued item: BBBBB" ) ;

            EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                      ""
                      EXCEPTION_NOT_EXPECTED ) ;
```

Figure 30-15 continued: ATH Test Script for Testing Task BUFFER with Timing Analysis

```
             begin

               BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
               delay 1.0;  -- Allow processing to be performed

             exception
             when others =>
                 ILLEGAL_EXCEPTION ( "Others" ) ;
             end ;

             DONE ;
           CHECK ("DEQUEUED VALUE", RESULT, "AAAAA");
           CHECK_TIMER ("SUN3_UNIX", 0.0000, 0.0005);

         END_TEST ;
-----------------------------------------------------------------------------
--      TEST CASE 4
-----------------------------------------------------------------------------
         START_TEST ( 4 ) ;
            -- COMMENT ( "Dequeue 2nd queued item: BBBBB" ) ;

             EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                       "",
                       EXCEPTION_NOT_EXPECTED ) ;

             begin

               BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
               delay 1.0;  -- Allow processing to be performed

             exception
             when others =>
                 ILLEGAL_EXCEPTION ( "Others" ) ;
             end ;

             DONE ;
           CHECK ("DEQUEUED VALUE", RESULT, "BBBBB");

         END_TEST ;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
-- CLOSE SCRIPT
----------------------------------------------------------------------------
----------------------------------------------------------------------------
       END_SCRIPT ;

end TEST_BUFFER ;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
-- End of main Test Script section
----------------------------------------------------------------------------
----------------------------------------------------------------------------
```

Figure 30-15 continued: ATH Test Script for Testing Task BUFFER with Timing Analysis

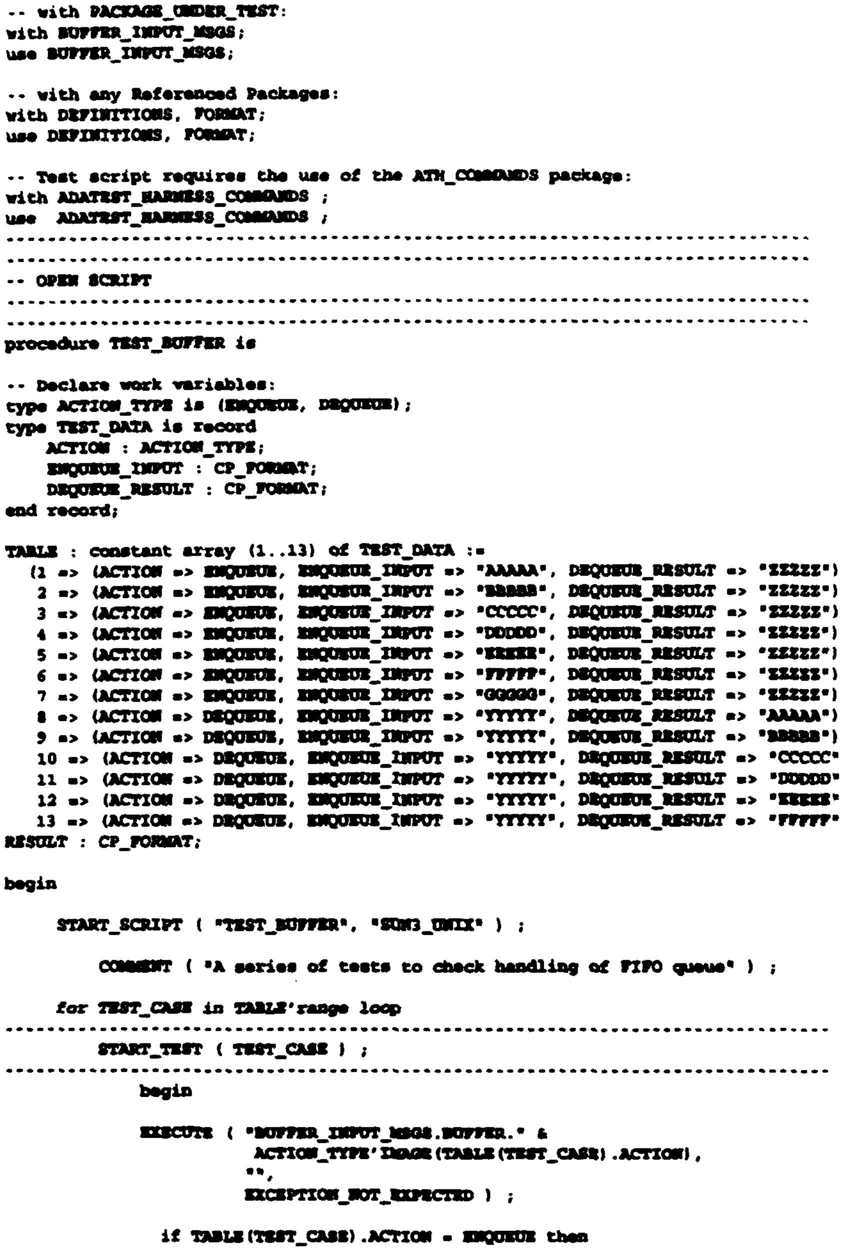


```
======================================================================================
        AdaTEST Harness        (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
--------------------------------------------------------------------------------------

        Test Results for       : TEST_BUFFER
        System Configured for : SUN3_UNIX
        Test Run on  19 MAY 1993  at  12:03:23

======================================================================================

START_SCRIPT: TEST_BUFFER , SUN3_UNIX ;

    -- A series of tests to check handling of FIFO queue
    RESET_TIMER ;

-------------------------------- TEST 1 --------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

        CHECK_TIMER: ( SUN3_UNIX ) ;PASSED
        TIMER_VALUE = 0.000 seconds.

------------------------------ END TEST 1 ------------------------------

-------------------------------- TEST 2 --------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------ END TEST 2 ------------------------------

    RESET_TIMER ;

-------------------------------- TEST 3 --------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item       "AAAAA"
        CHECK_TIMER: ( SUN3_UNIX ) ;PASSED
        TIMER_VALUE = 0.000 seconds.

------------------------------ END TEST 3 ------------------------------
```

Figure 30-16. ATH Results of Testing BUFFER with Timing Analysis

```
---------------------------------- TEST 4 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   **
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item       "BBBBB"
-------------------------------- END TEST 4 --------------------------------

END_SCRIPT: TEST_BUFFER ;

==============================================================================
        Test Results for TEST_BUFFER
        Test run completed at 12:03:27

        Script Errors            :  0
        Checks Passed            :  4
        Checks Failed            :  0
        Checks Ignored           :  0
        Paths with Stub Failures :  0
------------------------------------------------------------------------------
                   Overall Test PASSED
==============================================================================
```

Figure 30-16 continued: Results of Testing BUFFER with Timing Analysis

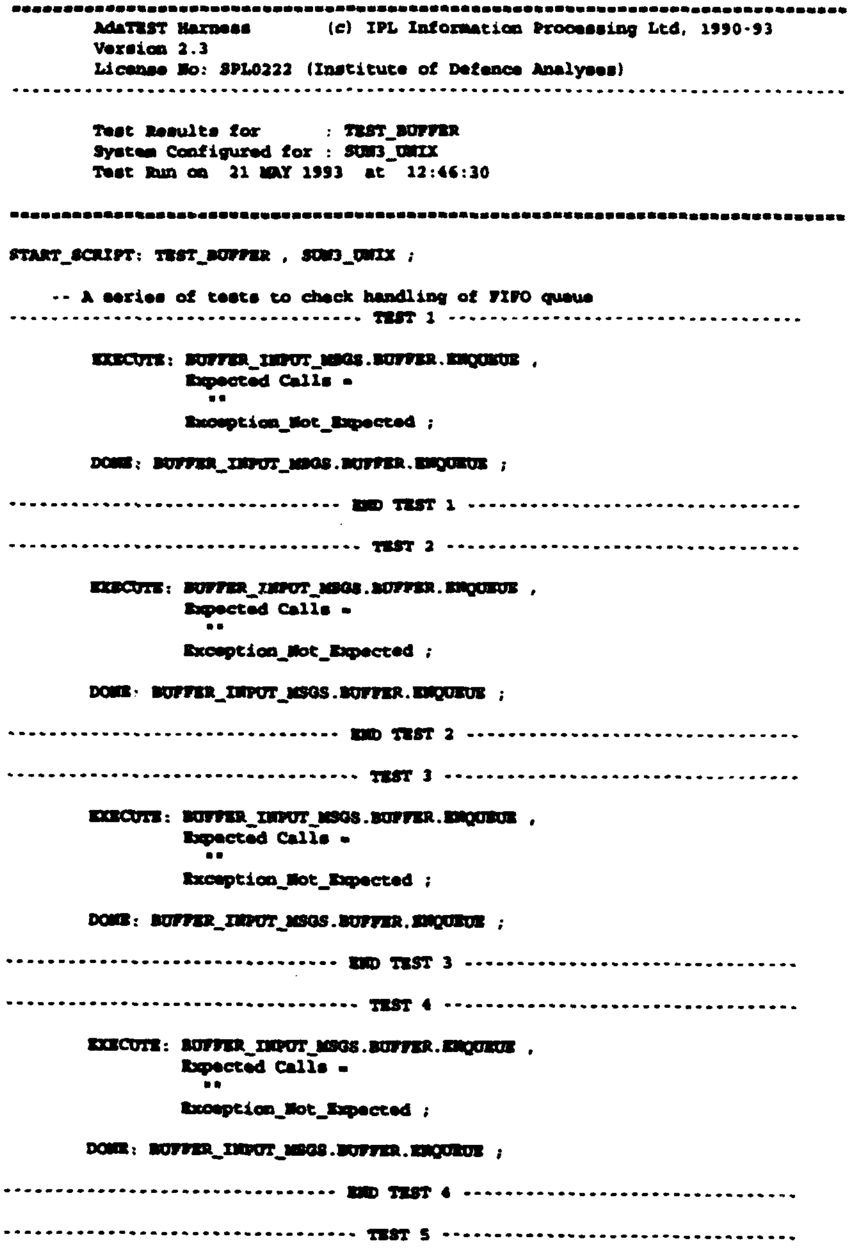


```
-- with PACKAGE_UNDER_TEST:
with BUFFER_INPUT_MSGS;
use BUFFER_INPUT_MSGS;

-- with any Referenced Packages:
with DEFINITIONS, FORMAT;
use DEFINITIONS, FORMAT;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-- OPEN SCRIPT
-----------------------------------------------------------------------
-----------------------------------------------------------------------
procedure TEST_BUFFER is

-- Declare work variables:
type ACTION_TYPE is (ENQUEUE, DEQUEUE);
type TEST_DATA is record
    ACTION : ACTION_TYPE;
    ENQUEUE_INPUT : CP_FORMAT;
    DEQUEUE_RESULT : CP_FORMAT;
end record;

TABLE : constant array (1..13) of TEST_DATA :=
  (1 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "AAAAA", DEQUEUE_RESULT => "ZZZZZ")
   2 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "BBBBB", DEQUEUE_RESULT => "ZZZZZ")
   3 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "CCCCC", DEQUEUE_RESULT => "ZZZZZ")
   4 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "DDDDD", DEQUEUE_RESULT => "ZZZZZ")
   5 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "EEEEE", DEQUEUE_RESULT => "ZZZZZ")
   6 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "FFFFF", DEQUEUE_RESULT => "ZZZZZ")
   7 => (ACTION => ENQUEUE, ENQUEUE_INPUT => "GGGGG", DEQUEUE_RESULT => "ZZZZZ")
   8 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "AAAAA")
   9 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "BBBBB")
   10 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "CCCCC"
   11 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "DDDDD"
   12 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "EEEEE"
   13 => (ACTION => DEQUEUE, ENQUEUE_INPUT => "YYYYY", DEQUEUE_RESULT => "FFFFF"
RESULT : CP_FORMAT;

begin

     START_SCRIPT ( "TEST_BUFFER", "SUN3_UNIX" ) ;

         COMMENT ( "A series of tests to check handling of FIFO queue" ) ;

     for TEST_CASE in TABLE'range loop
-------------------------------------------------------------------------
         START_TEST ( TEST_CASE ) ;
-------------------------------------------------------------------------
             begin

             EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER." &
                        ACTION_TYPE'IMAGE(TABLE(TEST_CASE).ACTION),
                       "",
                       EXCEPTION_NOT_EXPECTED ) ;

               if TABLE(TEST_CASE).ACTION = ENQUEUE then
```

Figure 30-17. ATH Test Script for Table-Driven Testing of BUFFER

```
            BUFFER_INPUT_MSGS.ENQUEUE (I => TABLE(TEST_CASE).ENQUEUE_INPUT)
          else
            BUFFER_INPUT_MSGS.DEQUEUE (RESULT);
          end if;
          delay 1.0;  -- Allow processing to be performed

        exception
        when others =>
            ILLEGAL_EXCEPTION ( "Others" ) ;
        end ;

        DONE ;

        if TABLE(TEST_CASE).ACTION = DEQUEUE then
          CHECK ("DEQUEUED VALUE", RESULT, TABLE(TEST_CASE).DEQUEUE_RESULT);
        end if;

    END_TEST ;
--------------------------------------------------------------------------
      end loop;
  END_SCRIPT;
end TEST_BUFFER;
--------------------------------------------------------------------------
```

Figure 30-17 continued: ATH Test Script for Table-Driven Testing of BUFFER



```
=============================================================================
        AdaTEST Harness         (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
-----------------------------------------------------------------------------

        Test Results for        : TEST_BUFFER
        System Configured for : SUN3_UNIX
        Test Run on  21 MAY 1993  at  12:46:30

=============================================================================

START_SCRIPT: TEST_BUFFER , SUN3_UNIX ;

    -- A series of tests to check handling of FIFO queue
-------------------------------- TEST 1 ---------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------ END TEST 1 -------------------------------

-------------------------------- TEST 2 ---------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------ END TEST 2 -------------------------------

-------------------------------- TEST 3 ---------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------ END TEST 3 -------------------------------

-------------------------------- TEST 4 ---------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------ END TEST 4 -------------------------------

-------------------------------- TEST 5 ---------------------------------
```

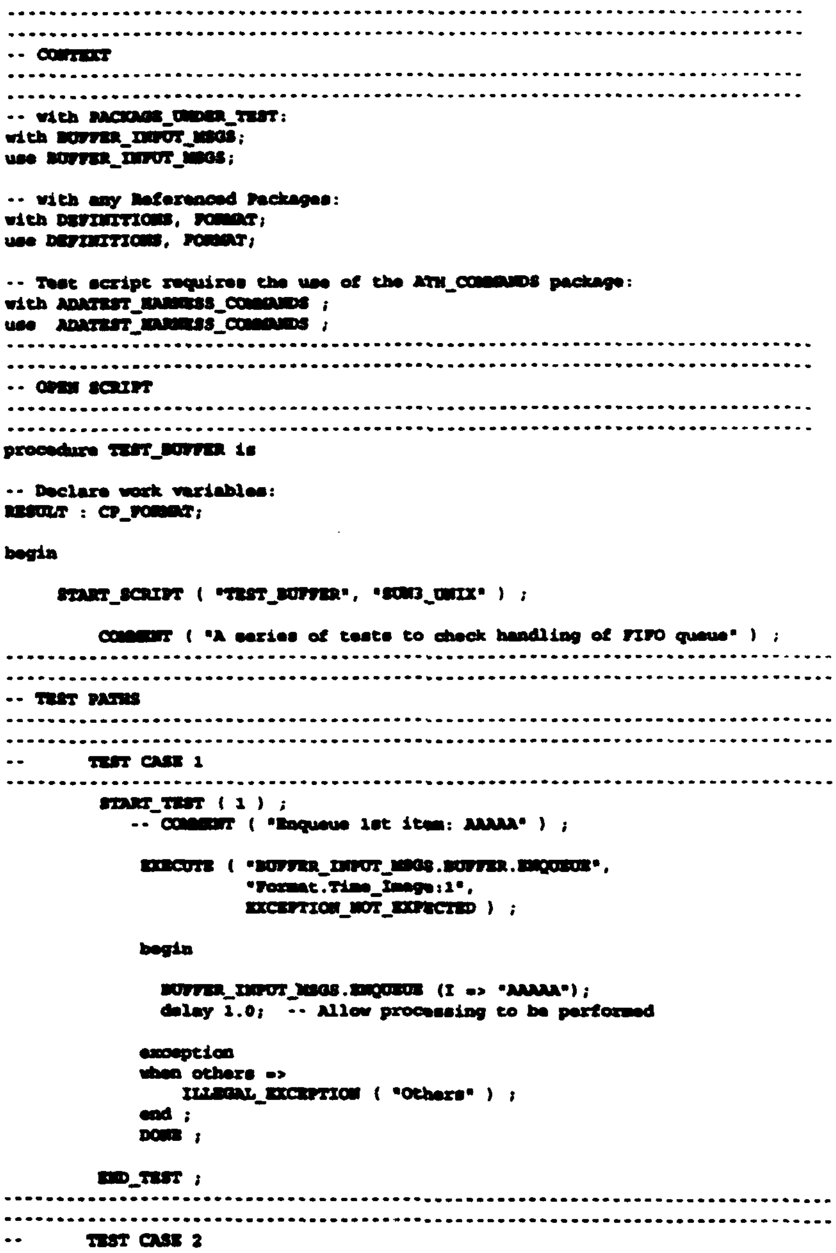
Figure 30-18. ATH Results of Table-Driven Testing of BUFFER

```
      EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
               Expected Calls =
                 ""
               Exception_Not_Expected ;

      DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------- END TEST 5 --------------------------------

--------------------------------- TEST 6 ----------------------------------

      EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
               Expected Calls =
                 ""
               Exception_Not_Expected ;

      DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------- END TEST 6 --------------------------------

--------------------------------- TEST 7 ----------------------------------

      EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
               Expected Calls =
                 ""
               Exception_Not_Expected ;

      DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

------------------------------- END TEST 7 --------------------------------

--------------------------------- TEST 8 ----------------------------------

      EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
               Expected Calls =
                 ""
               Exception_Not_Expected ;

      DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

      CHECK (  DEQUEUED VALUE ) ;PASSED
               Item       "AAAAA"
------------------------------- END TEST 8 --------------------------------

--------------------------------- TEST 9 ----------------------------------

      EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
               Expected Calls =
                 ""
               Exception_Not_Expected ;

      DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

      CHECK (  DEQUEUED VALUE ) ;PASSED
               Item       "BBBBB"
------------------------------- END TEST 9 --------------------------------

-- ------------------------------ TEST 10 ---------------------------------
```

Figure 30-18 continued: Results of Table-Driven Testing of BUFFER

```
        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "CCCCC"
-------------------------------- END TEST 10 --------------------------------

---------------------------------- TEST 11 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "DDDDD"
-------------------------------- END TEST 11 --------------------------------

---------------------------------- TEST 12 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "EEEEE"
-------------------------------- END TEST 12 --------------------------------

---------------------------------- TEST 13 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "FFFFF"
-------------------------------- END TEST 13 --------------------------------

END_SCRIPT: TEST_BUFFER ;

===============================================================================
        Test Results for TEST_BUFFER
        Test run completed at 12:46:44

        Script Errors            :  0
        Checks Passed            :  6
        Checks Failed            :  0
        Checks Ignored           :  0


        Paths with Stub Failures  :  0
-------------------------------------------------------------------------------
                   Overall Test PASSED
===============================================================================
```

Figure 30-18 continued: Results of Table-Driven Testing of BUFFER



```
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- CONTEXT
--------------------------------------------------------------------------
--------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with BUFFER_INPUT_MSGS;
use BUFFER_INPUT_MSGS;

-- with any Referenced Packages:
with DEFINITIONS, FORMAT;
use DEFINITIONS, FORMAT;

-- Test script requires the use of the ATH_COMMANDS package:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
---------------------------------------------------------------------------
---------------------------------------------------------------------------
-- OPEN SCRIPT
---------------------------------------------------------------------------
---------------------------------------------------------------------------
procedure TEST_BUFFER is

-- Declare work variables:
RESULT : CP_FORMAT;

begin

     START_SCRIPT ( "TEST_BUFFER", "SUN3_UNIX" ) ;

         COMMENT ( "A series of tests to check handling of FIFO queue" ) ;
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- TEST PATHS
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
--      TEST CASE 1
-----------------------------------------------------------------------------
         START_TEST ( 1 ) ;
            -- COMMENT ( "Enqueue 1st item: AAAAA" ) ;

             EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                       "Format.Time_Image:1",
                       EXCEPTION_NOT_EXPECTED ) ;

             begin

               BUFFER_INPUT_MSGS.ENQUEUE (I => "AAAAA");
               delay 1.0;  -- Allow processing to be performed

             exception
             when others =>
                 ILLEGAL_EXCEPTION ( "Others" ) ;
             end ;
             DONE ;

         END_TEST ;
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
--      TEST CASE 2
```

Figure 30-19. ATH Test Script for BUFFER with Sample Start Simulation

```
-----------------------------------------------------------------------------
        START_TEST ( 2 ) ;
           -- COMMENT ( "Enqueue 2nd item: BBBBB" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                     "Format.Time_Image:2",
                     EXCEPTION_NOT_EXPECTED ) ;

           begin

             BUFFER_INPUT_MSGS.ENQUEUE (I => "BBBBB");
             delay 1.0;  -- Allow processing to be performed

           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;

           DONE ;

        END_TEST ;
-----------------------------------------------------------------------------
--      TEST CASE 3
-----------------------------------------------------------------------------
        START_TEST ( 3 ) ;
           -- COMMENT ( "Dequeue 1st queued item: BBBBB" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                     "Format.Time_Image:3",
                     EXCEPTION_NOT_EXPECTED ) ;

           begin

             BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
             delay 1.0;  -- Allow processing to be performed

           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;

           DONE ;
         CHECK ("DEQUEUED VALUE", RESULT, "AAAAA");

        END_TEST ;
-----------------------------------------------------------------------------
--      TEST CASE 4
-----------------------------------------------------------------------------
        START_TEST ( 4 ) ;
           -- COMMENT ( "Dequeue 2nd queued item: BBBBB" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                     "Format.Time_Image:4",
                     EXCEPTION_NOT_EXPECTED ) ;

           begin

             BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
             delay 1.0;  -- Allow processing to be performed
```

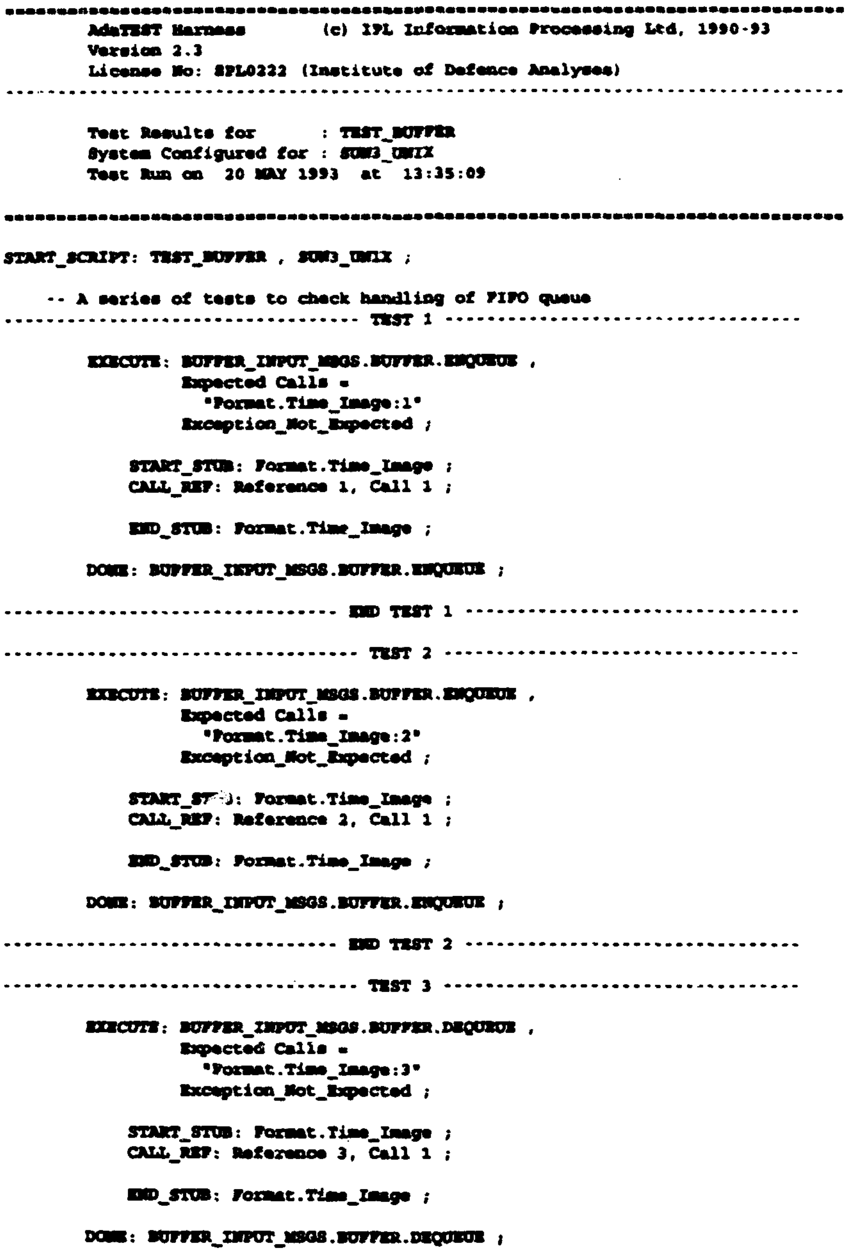
Figure 30-19 continued: ATH Test Script for BUFFER with Sample Start Simulation

```
            exception
            when others =>
                ILLEGAL_EXCEPTION ( "Others" ) ;
            end ;

            DONE ;
          CHECK ("DEQEUED VALUE", RESULT, "BBBBB");

        END_TEST ;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
-- CLOSE SCRIPT
----------------------------------------------------------------------------
----------------------------------------------------------------------------
       END_SCRIPT ;

end TEST_BUFFER ;
----------------------------------------------------------------------------
----------------------------------------------------------------------------
-- End of main Test Script section
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- STUB for Format.Time_Image.
-----------------------------------------------------------------------------
-- Stub simulation requires the use of the ATH Stub Simulation package:
with ADATEST_HARNESS_STUB_SIMULATION ;
use  ADATEST_HARNESS_STUB_SIMULATION ;

-- It may require use of the Commands or Generic Checks packages for Checks:
-- with ADATEST_HARNESS_COMMANDS ;
-- use  ADATEST_HARNESS_COMMANDS ;
-- with ADATEST_HARNESS_GENERIC_CHECKS ;
-- use  ADATEST_HARNESS_GENERIC_CHECKS ;

-- Stubs are coded as Separates, so provide the Package name:
separate (Format)

-- Declare the Stub's name and parameters:
function Time_Image (Clock_Time : in Calendar.Time) return STRING is

-- Instantiations of Generic_Checks if needed:

-- Declare work variables if any, and if stub is a function, declare a variable
-- of the correct type for returning set values:
RETURN_VALUE : STRING (1..20);

  begin

     START_STUB ("Format.Time_Image") ;

     case CALL_REF is
       when 1 =>
--               Check values of In parameters at point of call, if needed:

--               Set required Out parameter values  or value of return variable:
            RETURN_VALUE := "-- Enqueue time 1 --";
       when 2  =>
            RETURN_VALUE := "-- Enqueue time 2 --";
       when 3  =>
            RETURN_VALUE := "-- Dequeue time 1 --";
```

Figure 30-19 continued: ATH Test Script for BUFFER with Sample Start Simulation

```
        when 4   =>
             RETURN_VALUE := "-- Dequeue time 2 --";

-- At end of all clauses to deal with expected calls, provide the Illegal_Call
-- handler:
      when others =>
           ILLEGAL_CALL_REF ;
      end case ;

      END_STUB ;

-- If Stub is a function, return the return variable:
      return (RETURN_VALUE);

-- Exit the stub
end Time_Image;
--------------------------------------------------------------------------------
-- End of simulation section for Stub Format.Time_Image.
-------------------------------------------------------------------------------
```

Figure 30-19 continued ATH Test Script for BUFFER with Sample Start Simulation

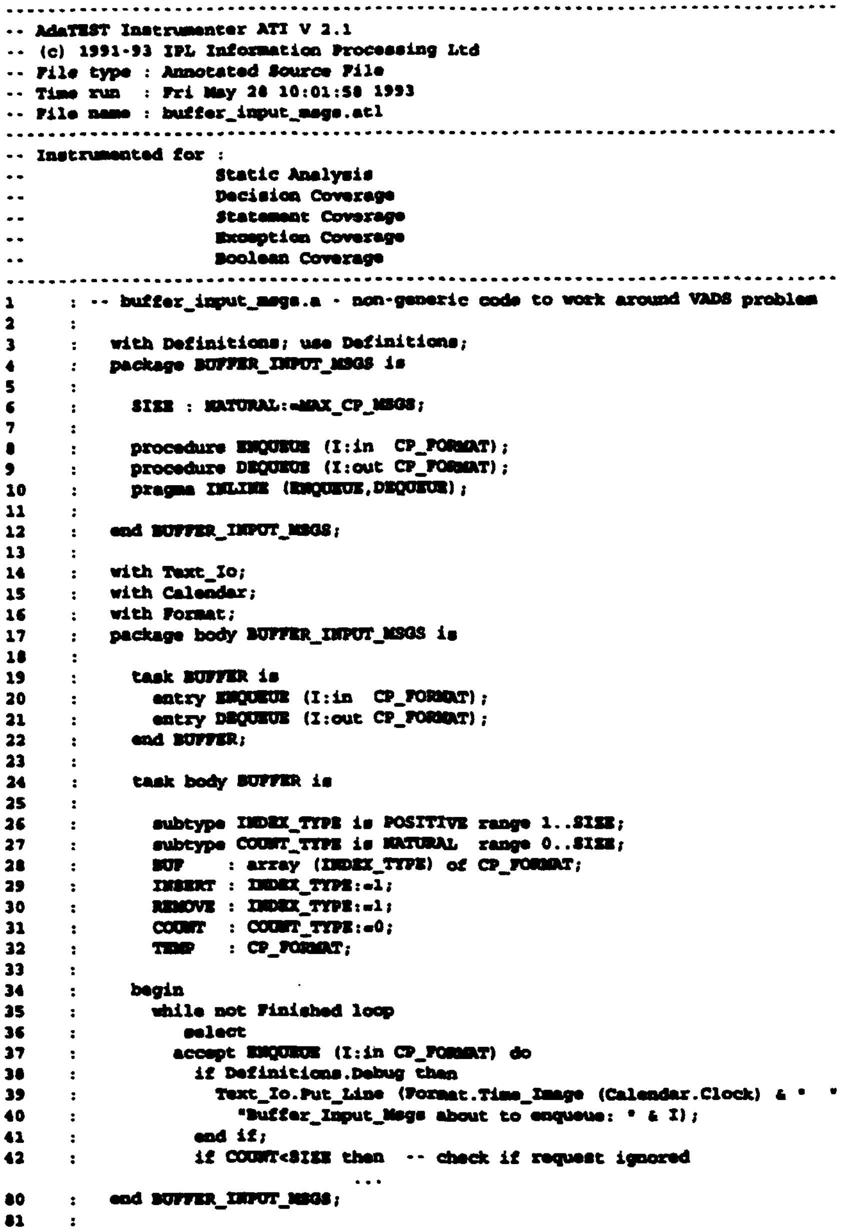


```
=========================================================================
        AdaTEST Harness         (c) IPL Information Processing Ltd, 1990-93
        Version 2.3
        License No: SPL0222 (Institute of Defence Analyses)
-------------------------------------------------------------------------

        Test Results for        : TEST_BUFFER
        System Configured for : SUN3_UNIX
        Test Run on  20 MAY 1993  at  13:35:09

=========================================================================

START_SCRIPT: TEST_BUFFER , SUN3_UNIX ;

    -- A series of tests to check handling of FIFO queue
---------------------------------- TEST 1 -------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   "Format.Time_Image:1"
                 Exception_Not_Expected ;

            START_STUB: Format.Time_Image ;
            CALL_REF: Reference 1, Call 1 ;

            END_STUB: Format.Time_Image ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

-------------------------------- END TEST 1 -----------------------------

---------------------------------- TEST 2 -------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   "Format.Time_Image:2"
                 Exception_Not_Expected ;

            START_STUB: Format.Time_Image ;
            CALL_REF: Reference 2, Call 1 ;

            END_STUB: Format.Time_Image ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

-------------------------------- END TEST 2 -----------------------------

---------------------------------- TEST 3 -------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   "Format.Time_Image:3"
                 Exception_Not_Expected ;

            START_STUB: Format.Time_Image ;
            CALL_REF: Reference 3, Call 1 ;

            END_STUB: Format.Time_Image ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;
```

Figure 30-20. ATH Results of Testing BUFFER with Sample Start Simulation

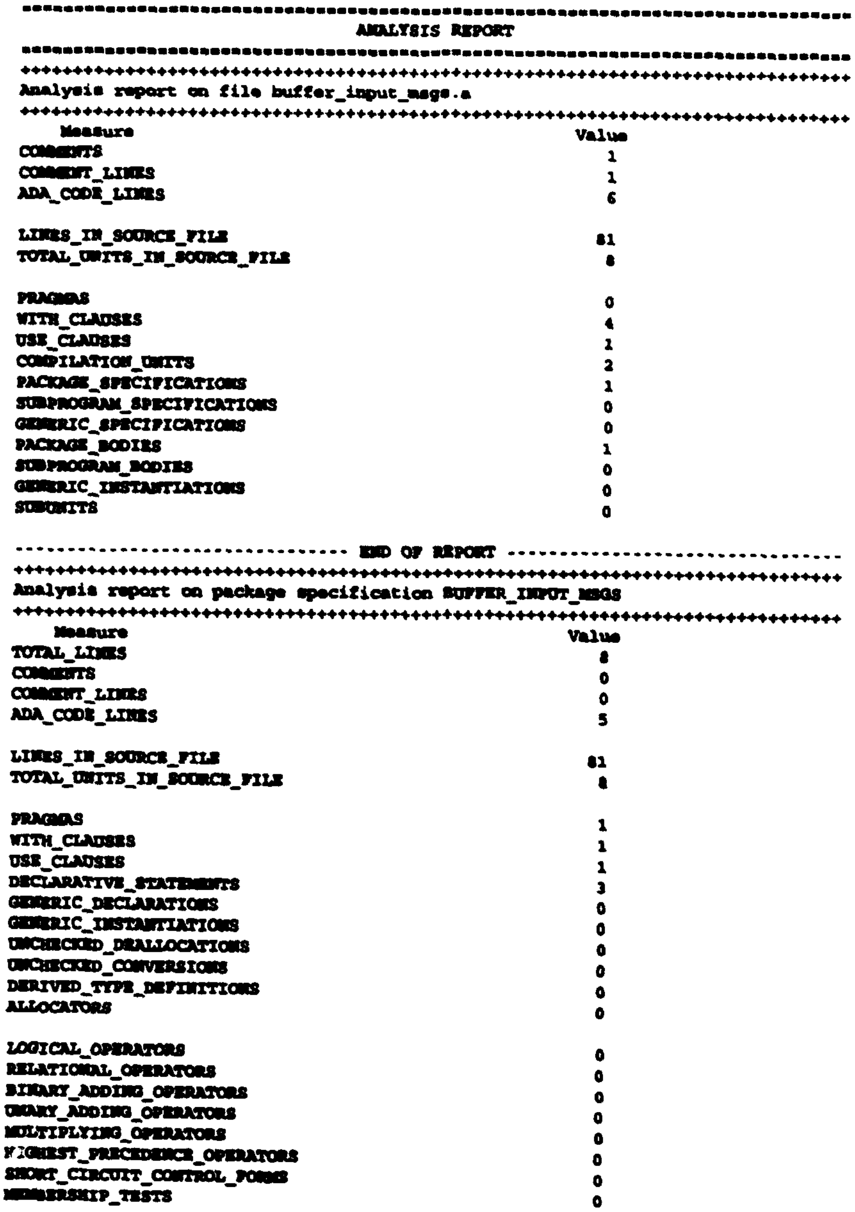
```
        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      'AAAAA'
------------------------------ END TEST 3 ----------------------------

------------------------------- TEST 4 -------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   "Format.Time_Image:4"
                 Exception_Not_Expected ;

            START_STUB: Format.Time_Image ;
            CALL_REF: Reference 4, Call 1 ;

            END_STUB: Format.Time_Image ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      'BBBBB'
------------------------------ END TEST 4 ----------------------------

END_SCRIPT: TEST_BUFFER ;

==============================================================================
        Test Results for TEST_BUFFER
        Test run completed at 13:35:13

        Script Errors             :  0
        Checks Passed             :  2
        Checks Failed             :  0
        Checks Ignored            :  0
        Paths with Stub Failures  :  0
------------------------------------------------------------------------------
                   Overall Test PASSED
==============================================================================
```

Figure 30-20 continued: Results of Testing BUFFER with Sample Start Simulation



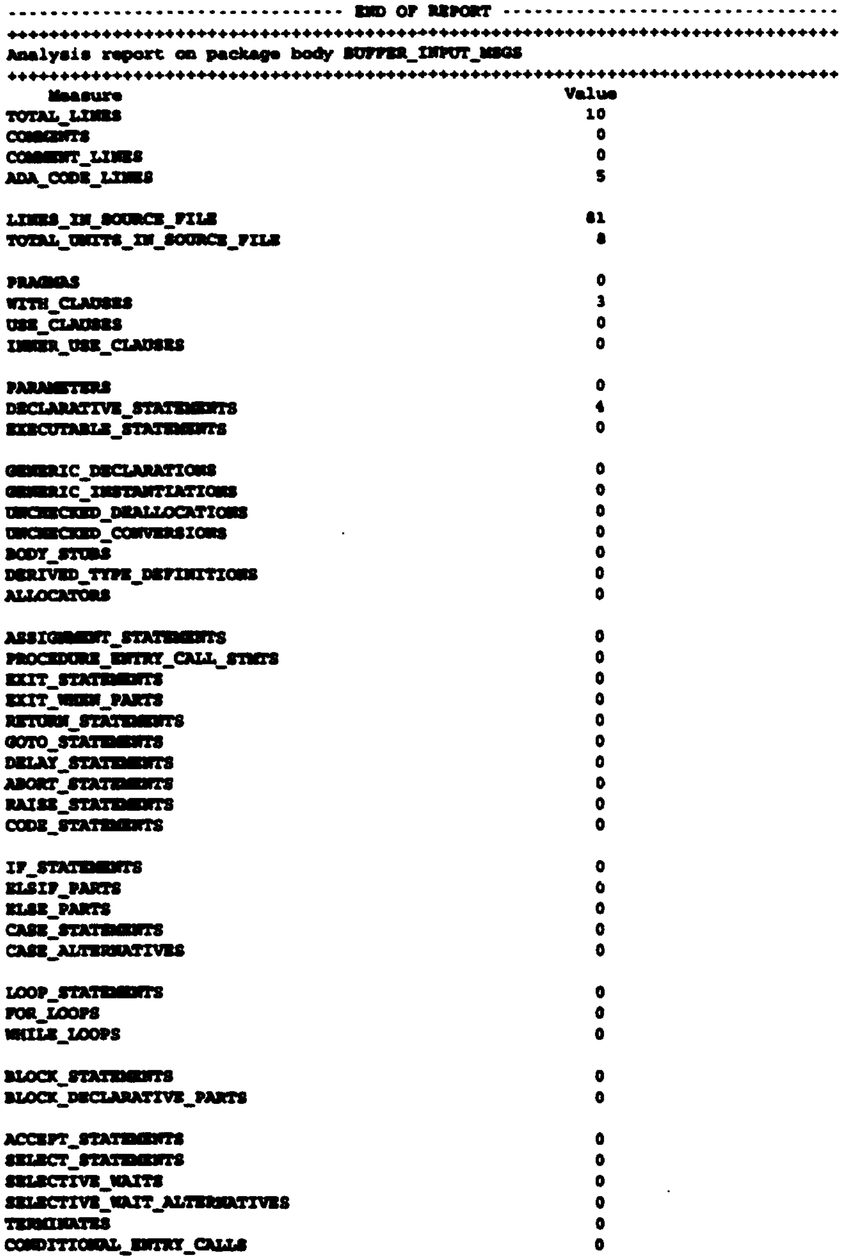
```
---------------------------------------------------------------------------
-- AdaTEST Instrumenter ATI V 2.1
-- (c) 1991-93 IPL Information Processing Ltd
-- File type : Annotated Source File
-- Time run  : Fri May 28 10:01:58 1993
-- File name : buffer_input_msgs.atl
---------------------------------------------------------------------------
-- Instrumented for :
--                   Static Analysis
--                   Decision Coverage
--                   Statement Coverage
--                   Exception Coverage
--                   Boolean Coverage
---------------------------------------------------------------------------
1     : -- buffer_input_msgs.a - non-generic code to work around VADS problem
2     :
3     :   with Definitions; use Definitions;
4     :   package BUFFER_INPUT_MSGS is
5     :
6     :     SIZE : NATURAL:=MAX_CP_MSGS;
7     :
8     :     procedure ENQUEUE (I:in  CP_FORMAT);
9     :     procedure DEQUEUE (I:out CP_FORMAT);
10    :     pragma INLINE (ENQUEUE,DEQUEUE);
11    :
12    :   end BUFFER_INPUT_MSGS;
13    :
14    :   with Text_Io;
15    :   with Calendar;
16    :   with Format;
17    :   package body BUFFER_INPUT_MSGS is
18    :
19    :     task BUFFER is
20    :       entry ENQUEUE (I:in  CP_FORMAT);
21    :       entry DEQUEUE (I:out CP_FORMAT);
22    :     end BUFFER;
23    :
24    :     task body BUFFER is
25    :
26    :       subtype INDEX_TYPE is POSITIVE range 1..SIZE;
27    :       subtype COUNT_TYPE is NATURAL  range 0..SIZE;
28    :       BUF    : array (INDEX_TYPE) of CP_FORMAT;
29    :       INSERT : INDEX_TYPE:=1;
30    :       REMOVE : INDEX_TYPE:=1;
31    :       COUNT  : COUNT_TYPE:=0;
32    :       TEMP   : CP_FORMAT;
33    :
34    :     begin
35    :       while not Finished loop
36    :          select
37    :         accept ENQUEUE (I:in CP_FORMAT) do
38    :           if Definitions.Debug then
39    :             Text_Io.Put_Line (Format.Time_Image (Calendar.Clock) & "  "
40    :               "Buffer_Input_Msgs about to enqueue: " & I);
41    :           end if;
42    :           if COUNT<SIZE then  -- check if request ignored
                                  ...
80    :   end BUFFER_INPUT_MSGS;
81    :
```

Figure 30-21. ATI Static Analysis of BUFFER Package



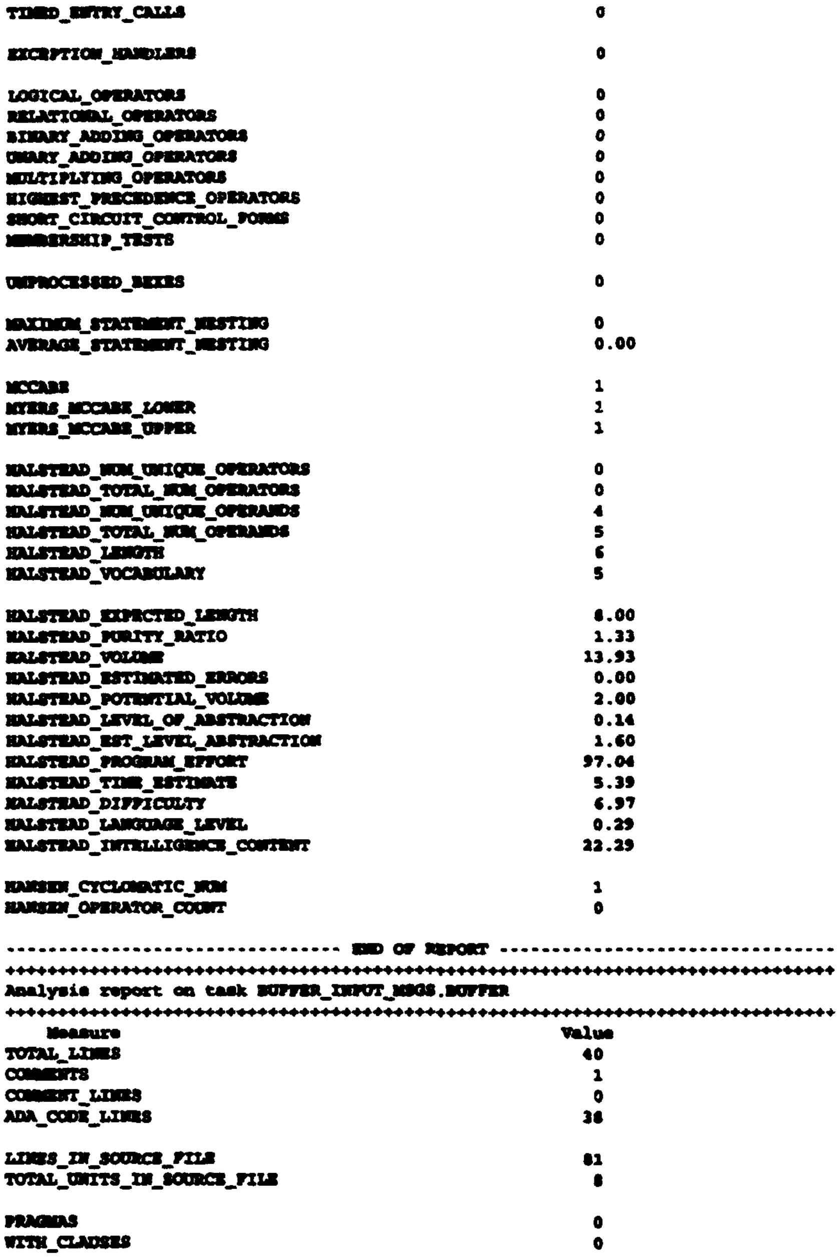
```
==============================================================================
                                ANALYSIS REPORT
==============================================================================
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on file buffer_input_msgs.a
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                                   Value
COMMENTS                                                        1
COMMENT_LINES                                                   1
ADA_CODE_LINES                                                  6

LINES_IN_SOURCE_FILE                                           81
TOTAL_UNITS_IN_SOURCE_FILE                                      8

PRAGMAS                                                         0
WITH_CLAUSES                                                    4
USE_CLAUSES                                                     1
COMPILATION_UNITS                                               2
PACKAGE_SPECIFICATIONS                                          1
SUBPROGRAM_SPECIFICATIONS                                       0
GENERIC_SPECIFICATIONS                                          0
PACKAGE_BODIES                                                  1
SUBPROGRAM_BODIES                                               0
GENERIC_INSTANTIATIONS                                          0
SUBUNITS                                                        0

------------------------------- END OF REPORT -------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on package specification BUFFER_INPUT_MSGS
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                                   Value
TOTAL_LINES                                                     8
COMMENTS                                                        0
COMMENT_LINES                                                   0
ADA_CODE_LINES                                                  5

LINES_IN_SOURCE_FILE                                           81
TOTAL_UNITS_IN_SOURCE_FILE                                      8

PRAGMAS                                                         1
WITH_CLAUSES                                                    1
USE_CLAUSES                                                     1
DECLARATIVE_STATEMENTS                                          3
GENERIC_DECLARATIONS                                            0
GENERIC_INSTANTIATIONS                                          0
UNCHECKED_DEALLOCATIONS                                         0
UNCHECKED_CONVERSIONS                                           0
DERIVED_TYPE_DEFINITIONS                                        0
ALLOCATORS                                                      0

LOGICAL_OPERATORS                                               0
RELATIONAL_OPERATORS                                            0
BINARY_ADDING_OPERATORS                                         0
UNARY_ADDING_OPERATORS                                          0
MULTIPLYING_OPERATORS                                           0
HIGHEST_PRECEDENCE_OPERATORS                                    0
SHORT_CIRCUIT_CONTROL_FORMS                                     0
MEMBERSHIP_TESTS                                                0
```

Figure 30-21 continued: ATI Static Analysis of BUFFER Package



```
-------------------------------- END OF REPORT -------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on package body BUFFER_INPUT_MSGS
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                            Value
TOTAL_LINES                                             10
COMMENTS                                                 0
COMMENT_LINES                                            0
ADA_CODE_LINES                                           5

LINES_IN_SOURCE_FILE                                    81
TOTAL_UNITS_IN_SOURCE_FILE                               8

PRAGMAS                                                  0
WITH_CLAUSES                                             3
USE_CLAUSES                                              0
INNER_USE_CLAUSES                                        0

PARAMETERS                                               0
DECLARATIVE_STATEMENTS                                   4
EXECUTABLE_STATEMENTS                                    0

GENERIC_DECLARATIONS                                     0
GENERIC_INSTANTIATIONS                                   0
UNCHECKED_DEALLOCATIONS                                  0
UNCHECKED_CONVERSIONS                                    0
BODY_STUBS                                               0
DERIVED_TYPE_DEFINITIONS                                 0
ALLOCATORS                                               0

ASSIGNMENT_STATEMENTS                                    0
PROCEDURE_ENTRY_CALL_STMTS                               0
EXIT_STATEMENTS                                          0
EXIT_WHEN_PARTS                                          0
RETURN_STATEMENTS                                        0
GOTO_STATEMENTS                                          0
DELAY_STATEMENTS                                         0
ABORT_STATEMENTS                                         0
RAISE_STATEMENTS                                         0
CODE_STATEMENTS                                          0

IF_STATEMENTS                                            0
ELSIF_PARTS                                              0
ELSE_PARTS                                               0
CASE_STATEMENTS                                          0
CASE_ALTERNATIVES                                        0

LOOP_STATEMENTS                                          0
FOR_LOOPS                                                0
WHILE_LOOPS                                              0

BLOCK_STATEMENTS                                         0
BLOCK_DECLARATIVE_PARTS                                  0

ACCEPT_STATEMENTS                                        0
SELECT_STATEMENTS                                        0
SELECTIVE_WAITS                                          0
SELECTIVE_WAIT_ALTERNATIVES                              0
TERMINATES                                               0
CONDITIONAL_ENTRY_CALLS                                  0
```

Figure 30-21 continued: ATI Static Analysis of BUFFER Package

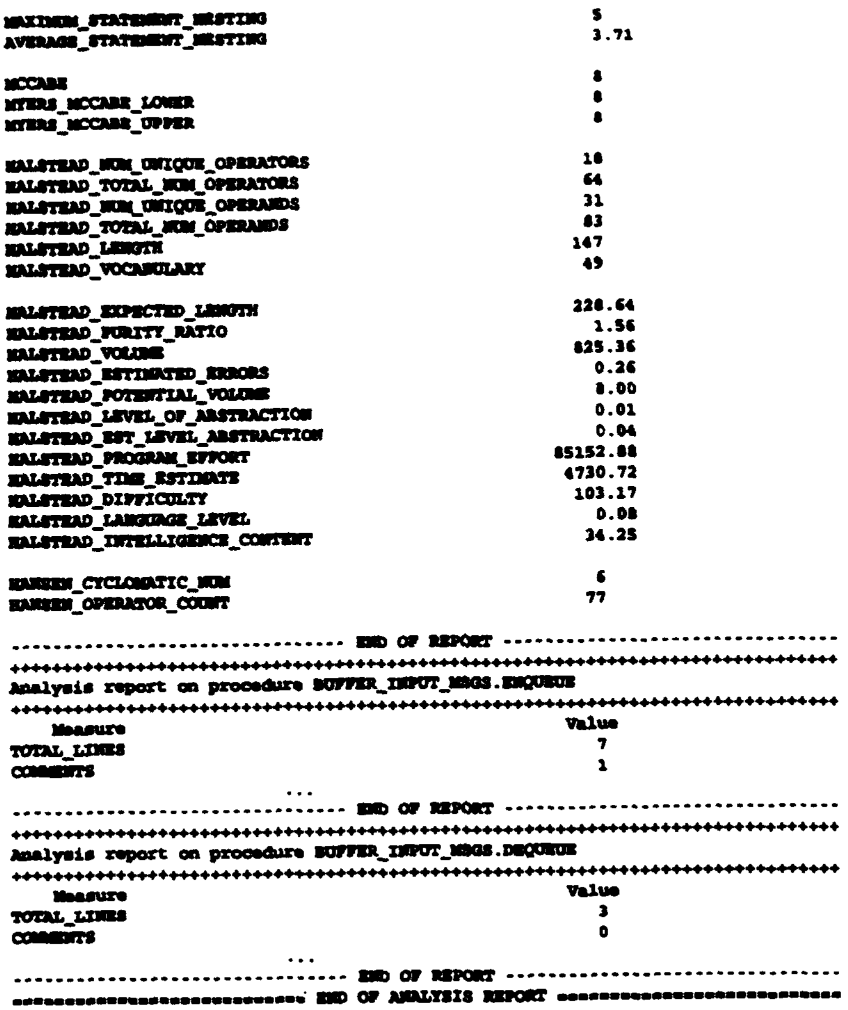


```
TIMED_ENTRY_CALLS                              0

EXCEPTION_HANDLERS                             0

LOGICAL_OPERATORS                              0
RELATIONAL_OPERATORS                           0
BINARY_ADDING_OPERATORS                        0
UNARY_ADDING_OPERATORS                         0
MULTIPLYING_OPERATORS                          0
HIGHEST_PRECEDENCE_OPERATORS                   0
SHORT_CIRCUIT_CONTROL_FORMS                    0
MEMBERSHIP_TESTS                               0

UNPROCESSED_BEXES                              0

MAXIMUM_STATEMENT_NESTING                      0
AVERAGE_STATEMENT_NESTING                      0.00

MCCABE                                         1
MYERS_MCCABE_LOWER                             1
MYERS_MCCABE_UPPER                             1

HALSTEAD_NUM_UNIQUE_OPERATORS                  0
HALSTEAD_TOTAL_NUM_OPERATORS                   0
HALSTEAD_NUM_UNIQUE_OPERANDS                   4
HALSTEAD_TOTAL_NUM_OPERANDS                    5
HALSTEAD_LENGTH                                6
HALSTEAD_VOCABULARY                            5

HALSTEAD_EXPECTED_LENGTH                       8.00
HALSTEAD_PURITY_RATIO                          1.33
HALSTEAD_VOLUME                               13.93
HALSTEAD_ESTIMATED_ERRORS                      0.00
HALSTEAD_POTENTIAL_VOLUME                      2.00
HALSTEAD_LEVEL_OF_ABSTRACTION                  0.14
HALSTEAD_EST_LEVEL_ABSTRACTION                 1.60
HALSTEAD_PROGRAM_EFFORT                       97.04
HALSTEAD_TIME_ESTIMATE                         5.39
HALSTEAD_DIFFICULTY                            6.97
HALSTEAD_LANGUAGE_LEVEL                        0.29
HALSTEAD_INTELLIGENCE_CONTENT                 22.29

HANSEN_CYCLOMATIC_NUM                          1
HANSEN_OPERATOR_COUNT                          0

-------------------------------- END OF REPORT --------------------------------
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on task BUFFER_INPUT_MSGS.BUFFER
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                  Value
TOTAL_LINES                                    40
COMMENTS                                       1
COMMENT_LINES                                  0
ADA_CODE_LINES                                 38

LINES_IN_SOURCE_FILE                           81
TOTAL_UNITS_IN_SOURCE_FILE                     8

PRAGMAS                                        0
WITH_CLAUSES                                   0
```

Figure 30-21 continued: ATI Static Analysis of BUFFER Package

```
USE_CLAUSES                                        0
INNER_USE_CLAUSES                                  0

PARAMETERS                                         2
DECLARATIVE_STATEMENTS                             7
EXECUTABLE_STATEMENTS                             17

GENERIC_DECLARATIONS                               0
GENERIC_INSTANTIATIONS                             0
UNCHECKED_DEALLOCATIONS                            0
UNCHECKED_CONVERSIONS                              0
BODY_STUBS                                         0
DERIVED_TYPE_DEFINITIONS                           0
ALLOCATORS                                         0

ASSIGNMENT_STATEMENTS                              7
PROCEDURE_ENTRY_CALL_STMTS                         2
EXIT_STATEMENTS                                    0
EXIT_WHEN_PARTS                                    0
RETURN_STATEMENTS                                  0
GOTO_STATEMENTS                                    0
DELAY_STATEMENTS                                   1
ABORT_STATEMENTS                                   0
RAISE_STATEMENTS                                   0
CODE_STATEMENTS                                    0

IF_STATEMENTS                                      3
ELSIF_PARTS                                        0
ELSE_PARTS                                         0
CASE_STATEMENTS                                    0
CASE_ALTERNATIVES                                  0

LOOP_STATEMENTS                                    1
FOR_LOOPS                                          0
WHILE_LOOPS                                        1

BLOCK_STATEMENTS                                   0
BLOCK_DECLARATIVE_PARTS                            0

ACCEPT_STATEMENTS                                  2
SELECT_STATEMENTS                                  1
SELECTIVE_WAITS                                    1
SELECTIVE_WAIT_ALTERNATIVES                        3
TERMINATES                                         0
CONDITIONAL_ENTRY_CALLS                            0
TIMED_ENTRY_CALLS                                  0

EXCEPTION_HANDLERS                                 0

LOGICAL_OPERATORS                                  0
RELATIONAL_OPERATORS                               2
BINARY_ADDING_OPERATORS                           10
UNARY_ADDING_OPERATORS                             0
MULTIPLYING_OPERATORS                              2
HIGHEST_PRECEDENCE_OPERATORS                       1
SHORT_CIRCUIT_CONTROL_FORMS                        0
MEMBERSHIP_TESTS                                   0

UNPROCESSED_BEXES                                  0
```

Figure 30-21 continued: ATI Static Analysis of BUFFER Package



```
MAXIMUM_STATEMENT_NESTING                                   5
AVERAGE_STATEMENT_NESTING                                   3.71

MCCABE                                                      8
MYERS_MCCABE_LOWER                                          8
MYERS_MCCABE_UPPER                                          8

HALSTEAD_NUM_UNIQUE_OPERATORS                              18
HALSTEAD_TOTAL_NUM_OPERATORS                               64
HALSTEAD_NUM_UNIQUE_OPERANDS                               31
HALSTEAD_TOTAL_NUM_OPERANDS                                83
HALSTEAD_LENGTH                                           147
HALSTEAD_VOCABULARY                                        49

HALSTEAD_EXPECTED_LENGTH                                  228.64
HALSTEAD_PURITY_RATIO                                       1.56
HALSTEAD_VOLUME                                           825.36
HALSTEAD_ESTIMATED_ERRORS                                   0.26
HALSTEAD_POTENTIAL_VOLUME                                   8.00
HALSTEAD_LEVEL_OF_ABSTRACTION                               0.01
HALSTEAD_EST_LEVEL_ABSTRACTION                              0.04
HALSTEAD_PROGRAM_EFFORT                                 85152.88
HALSTEAD_TIME_ESTIMATE                                   4730.72
HALSTEAD_DIFFICULTY                                       103.17
HALSTEAD_LANGUAGE_LEVEL                                     0.08
HALSTEAD_INTELLIGENCE_CONTENT                              34.25

HANSEN_CYCLOMATIC_NUM                                       6
HANSEN_OPERATOR_COUNT                                      77
-------------------------------- END OF REPORT --------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on procedure BUFFER_INPUT_MSGS.ENQUEUE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                               Value
TOTAL_LINES                                                 7
COMMENTS                                                    1
                              ...
-------------------------------- END OF REPORT --------------------------------
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis report on procedure BUFFER_INPUT_MSGS.DEQUEUE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Measure                                               Value
TOTAL_LINES                                                 3
COMMENTS                                                    0
                              ...
-------------------------------- END OF REPORT --------------------------------
============================ END OF ANALYSIS REPORT ============================
```

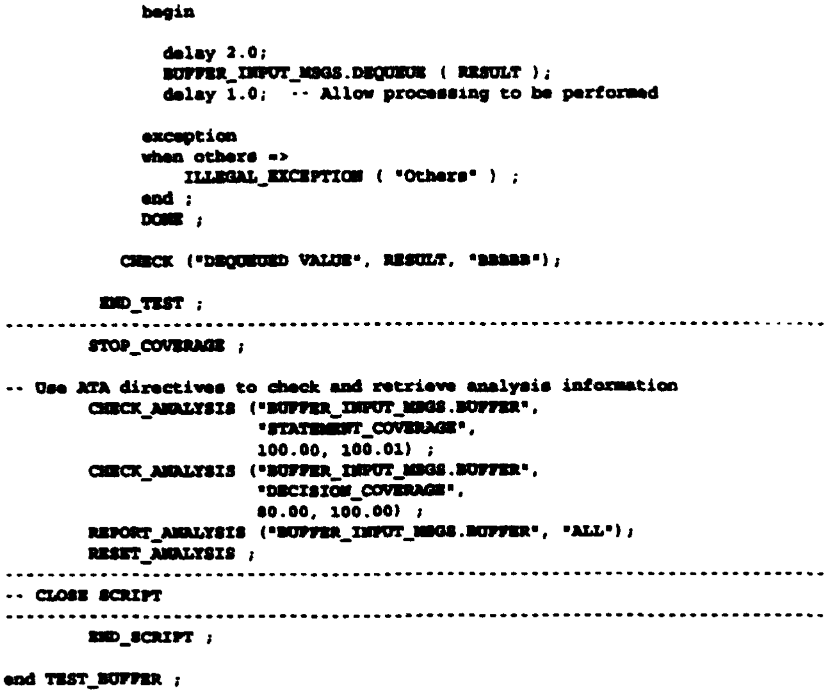
Figure 30-21 continued: ATI Static Analysis of BUFFER Package

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- CONTEXT
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-- with PACKAGE_UNDER_TEST:
with BUFFER_INPUT_MSGS;
use BUFFER_INPUT_MSGS;

-- with any Referenced Packages:
with DEFINITIONS, FORMAT;
use DEFINITIONS, FORMAT;

-- Test script requires the use of the ATH_COMMANDS and ANALYSIS packages:
with ADATEST_HARNESS_COMMANDS ;
use  ADATEST_HARNESS_COMMANDS ;
with ADATEST_HARNESS_ANALYSIS ;
use  ADATEST_HARNESS_ANALYSIS ;
---------------------------------------------------------------------------------
-- OPEN SCRIPT
---------------------------------------------------------------------------------
procedure TEST_BUFFER is

-- Instantiations of Generic_Checks if needed:

-- Declare work variables:
RESULT : CP_FORMAT;

begin

     START_SCRIPT ( "TEST_BUFFER", "SUN3_UNIX" ) ;
     INITIALISE_ANALYSIS ;
     SET_TRACE ( TRACE_FULL ) ;
     START_COVERAGE ;

         COMMENT ( "A series of tests to check handling of FIFO queue" ) ;
----------------------------------------------------------------------------------
-- TEST PATHS
----------------------------------------------------------------------------------
--       TEST CASE 1
----------------------------------------------------------------------------------
         START_TEST ( 1 ) ;
            -- COMMENT ( "Enqueue 1st item: AAAAA" ) ;
             EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                       "",
                       EXCEPTION_NOT_EXPECTED ) ;

             begin

               delay 2.0;
               BUFFER_INPUT_MSGS.ENQUEUE (I => "AAAAA");
               delay 1.0;  -- Allow processing to be performed

             exception
             when others =>
                 ILLEGAL_EXCEPTION ( "Others" ) ;
             end ;
             DONE ;

         END_TEST ;
```

Figure 30-22. ATA Test Script for Coverage and Trace Analysis of BUFFER Package

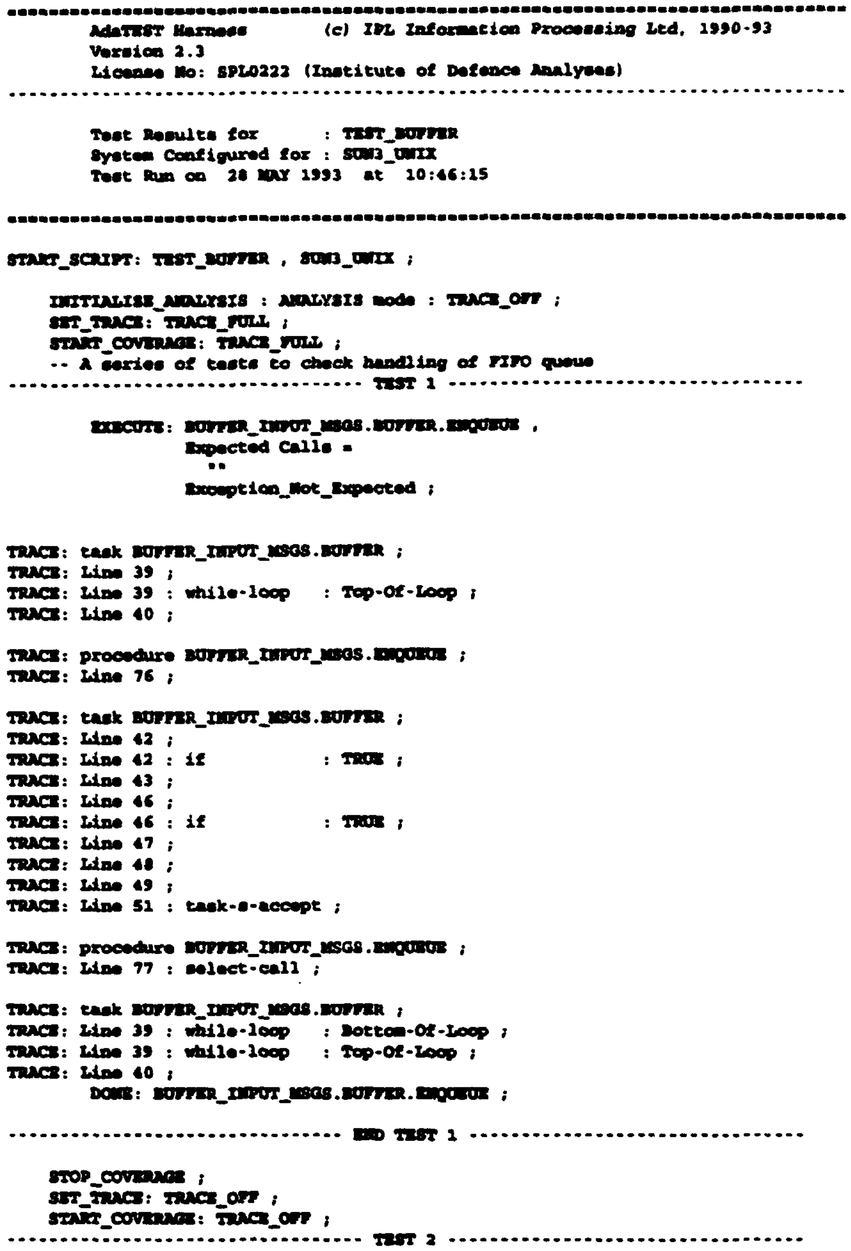
```
STOP_COVERAGE ;
SET_TRACE ( TRACE_OFF ) ;
START_COVERAGE ;
-----------------------------------------------------------------------------
--      TEST CASE 2
-----------------------------------------------------------------------------
        START_TEST ( 2 ) ;
          -- COMMENT ( "Enqueue 2nd item: BBBBB" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.ENQUEUE",
                     "",
                     EXCEPTION_NOT_EXPECTED ) ;

           begin

             delay 2.0;
             BUFFER_INPUT_MSGS.ENQUEUE (I => "BBBBB");
             delay 1.0;  -- Allow processing to be performed

           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;
           DONE ;

        END_TEST ;
-----------------------------------------------------------------------------
--      TEST CASE 3
-----------------------------------------------------------------------------
        START_TEST ( 3 ) ;
          -- COMMENT ( "Dequeue 1st queued item: AAAAA" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                     "",
                     EXCEPTION_NOT_EXPECTED ) ;

           begin

             delay 2.0;
             BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
             delay 1.0;  -- Allow processing to be performed

           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;
           DONE ;

          CHECK ("DEQUEUED VALUE", RESULT, "AAAAA");

        END_TEST ;
-----------------------------------------------------------------------------
--      TEST CASE 4
-----------------------------------------------------------------------------
        START_TEST ( 4 ) ;
          -- COMMENT ( "Dequeue 2nd queued item: BBBBB" ) ;

           EXECUTE ( "BUFFER_INPUT_MSGS.BUFFER.DEQUEUE",
                     "",
                     EXCEPTION_NOT_EXPECTED ) ;
```

Figure 30-22 continued: ATA Test Script for Coverage and Trace Analysis of BUFFER Package



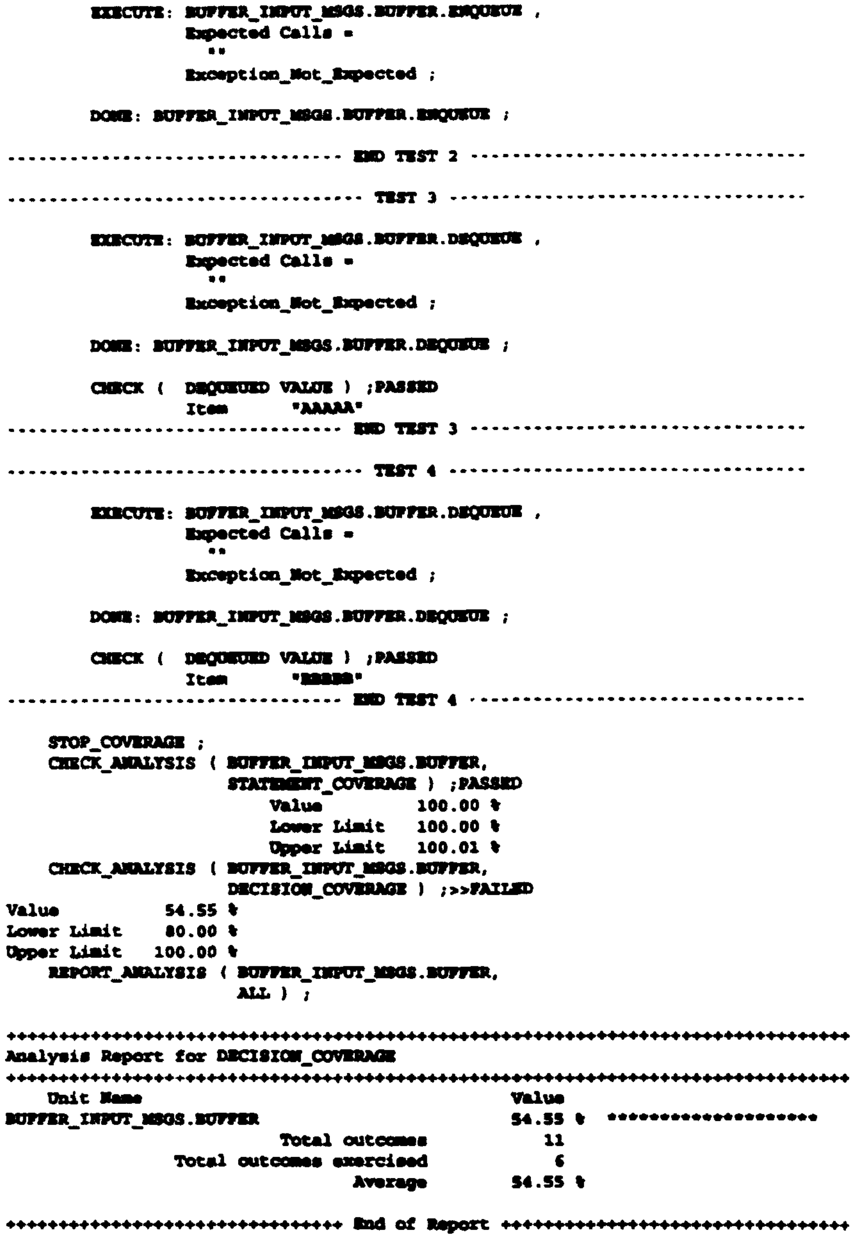
```
           begin

             delay 2.0;
             BUFFER_INPUT_MSGS.DEQUEUE ( RESULT );
             delay 1.0;  -- Allow processing to be performed

           exception
           when others =>
               ILLEGAL_EXCEPTION ( "Others" ) ;
           end ;
           DONE ;

         CHECK ("DEQUEUED VALUE", RESULT, "BBBBB");

       END_TEST ;
------------------------------------------------------------------------
      STOP_COVERAGE ;

-- Use ATA directives to check and retrieve analysis information
      CHECK_ANALYSIS ("BUFFER_INPUT_MSGS.BUFFER",
                      "STATEMENT_COVERAGE",
                      100.00, 100.01) ;
      CHECK_ANALYSIS ("BUFFER_INPUT_MSGS.BUFFER",
                      "DECISION_COVERAGE",
                      80.00, 100.00) ;
      REPORT_ANALYSIS ("BUFFER_INPUT_MSGS.BUFFER", "ALL");
      RESET_ANALYSIS ;
------------------------------------------------------------------------
-- CLOSE SCRIPT
------------------------------------------------------------------------
      END_SCRIPT ;

end TEST_BUFFER ;
```

Figure 30-22 continued: ATA Test Script for Coverage and Trace Analysis of BUFFER Package



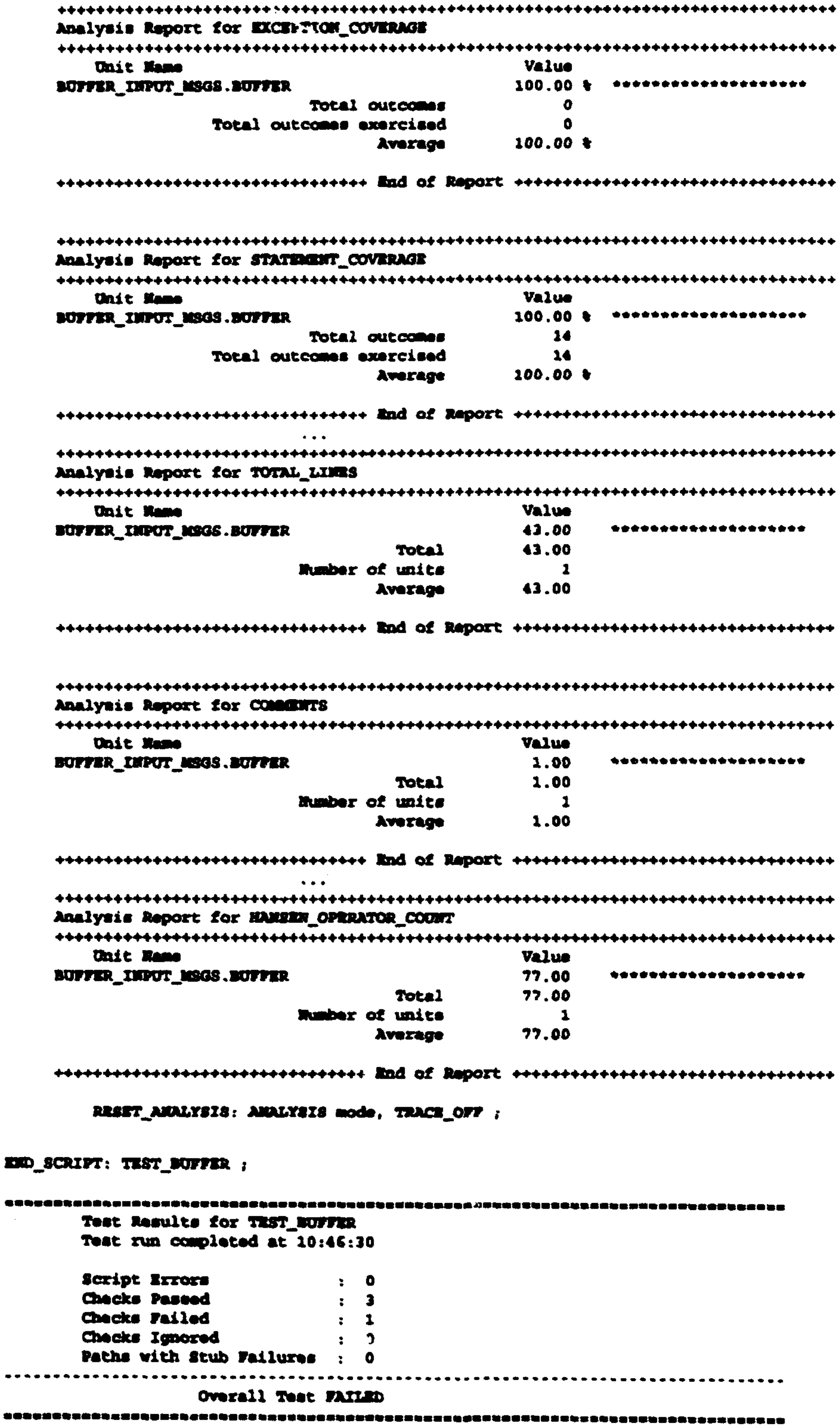
```
========================================================================
       AdaTEST Harness        (c) IPL Information Processing Ltd, 1990-93
       Version 2.3
       License No: SPL0222 (Institute of Defence Analyses)
------------------------------------------------------------------------

       Test Results for       : TEST_BUFFER
       System Configured for : SUN3_UNIX
       Test Run on  28 MAY 1993  at  10:46:15

========================================================================

START_SCRIPT: TEST_BUFFER , SUN3_UNIX ;

   INITIALISE_ANALYSIS : ANALYSIS mode : TRACE_OFF ;
   SET_TRACE: TRACE_FULL ;
   START_COVERAGE: TRACE_FULL ;
   -- A series of tests to check handling of FIFO queue
------------------------------ TEST 1 ------------------------------

       EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                Expected Calls =
                  ""
                Exception_Not_Expected ;


TRACE: task BUFFER_INPUT_MSGS.BUFFER ;
TRACE: Line 39 ;
TRACE: Line 39 : while-loop    : Top-Of-Loop ;
TRACE: Line 40 ;

TRACE: procedure BUFFER_INPUT_MSGS.ENQUEUE ;
TRACE: Line 76 ;

TRACE: task BUFFER_INPUT_MSGS.BUFFER ;
TRACE: Line 42 ;
TRACE: Line 42 : if            : TRUE ;
TRACE: Line 43 ;
TRACE: Line 46 ;
TRACE: Line 46 : if            : TRUE ;
TRACE: Line 47 ;
TRACE: Line 48 ;
TRACE: Line 49 ;
TRACE: Line 51 : task-s-accept ;

TRACE: procedure BUFFER_INPUT_MSGS.ENQUEUE ;
TRACE: Line 77 : select-call ;

TRACE: task BUFFER_INPUT_MSGS.BUFFER ;
TRACE: Line 39 : while-loop    : Bottom-Of-Loop ;
TRACE: Line 39 : while-loop    : Top-Of-Loop ;
TRACE: Line 40 ;
       DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

---------------------------- END TEST 1 ----------------------------

   STOP_COVERAGE ;
   SET_TRACE: TRACE_OFF ;
   START_COVERAGE: TRACE_OFF ;
------------------------------ TEST 2 ------------------------------
```

Figure 30-23. ATA Results for Coverage and Trace Analysis of BUFFER Package



```
        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.ENQUEUE ;

-------------------------------- END TEST 2 --------------------------------

---------------------------------- TEST 3 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "AAAAA"
-------------------------------- END TEST 3 --------------------------------

---------------------------------- TEST 4 ----------------------------------

        EXECUTE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ,
                 Expected Calls =
                   ""
                 Exception_Not_Expected ;

        DONE: BUFFER_INPUT_MSGS.BUFFER.DEQUEUE ;

        CHECK (  DEQUEUED VALUE ) ;PASSED
                 Item      "BBBBB"
-------------------------------- END TEST 4 --------------------------------

    STOP_COVERAGE ;
    CHECK_ANALYSIS ( BUFFER_INPUT_MSGS.BUFFER,
                     STATEMENT_COVERAGE ) ;PASSED
                         Value          100.00 %
                         Lower Limit    100.00 %
                         Upper Limit    100.01 %
    CHECK_ANALYSIS ( BUFFER_INPUT_MSGS.BUFFER,
                     DECISION_COVERAGE ) ;>>FAILED
Value          54.55 %
Lower Limit    80.00 %
Upper Limit   100.00 %
    REPORT_ANALYSIS ( BUFFER_INPUT_MSGS.BUFFER,
                      ALL ) ;

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for DECISION_COVERAGE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                       Value
BUFFER_INPUT_MSGS.BUFFER                            54.55 %  ********************
                           Total outcomes             11
                 Total outcomes exercised              6
                                  Average           54.55 %

+++++++++++++++++++++++++++++++++ End of Report +++++++++++++++++++++++++++++++++
```

Figure 30-23 continued: ATA Results for Coverage and Trace Analysis of BUFFER Package



```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for EXCEPTION_COVERAGE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                         Value
BUFFER_INPUT_MSGS.BUFFER                             100.00 %  *******************
                        Total outcomes                  0
                Total outcomes exercised                0
                                 Average             100.00 %

+++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for STATEMENT_COVERAGE
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                         Value
BUFFER_INPUT_MSGS.BUFFER                             100.00 %  *******************
                        Total outcomes                 14
                Total outcomes exercised               14
                                 Average             100.00 %

+++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++
                        ...
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for TOTAL_LINES
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                         Value
BUFFER_INPUT_MSGS.BUFFER                              43.00    *******************
                                   Total              43.00
                         Number of units                1
                                 Average              43.00

+++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++


++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for COMMENTS
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                         Value
BUFFER_INPUT_MSGS.BUFFER                               1.00    *******************
                                   Total               1.00
                         Number of units                1
                                 Average               1.00

+++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++
                        ...
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Analysis Report for HANSEN_OPERATOR_COUNT
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Unit Name                                         Value
BUFFER_INPUT_MSGS.BUFFER                              77.00    *******************
                                   Total              77.00
                         Number of units                1
                                 Average              77.00

+++++++++++++++++++++++++++++++ End of Report ++++++++++++++++++++++++++++++++

    RESET_ANALYSIS: ANALYSIS mode, TRACE_OFF ;

END_SCRIPT: TEST_BUFFER ;

===============================================================================
       Test Results for TEST_BUFFER
       Test run completed at 10:46:30

       Script Errors             :  0
       Checks Passed             :  3
       Checks Failed             :  1
       Checks Ignored            :  0
       Paths with Stub Failures  :  0
-------------------------------------------------------------------------------
                   Overall Test FAILED
===============================================================================
```

Figure 30-23 continued: ATA Results for Coverage and Trace Analysis of BUFFER Package

31. METRIC

METRIC is the newest member of Software Research's Software Testworks (STW) toolset (see IDA Paper-2769, pp. 25-1 to 25-35). It provides static code analysis and is intended to support maintenance activities by computing several complexity metrics. METRIC supports two of the most popular families of metrics: Software Science and Cyclomatic Complexity.

31.1 Tool Overview

METRIC was developed by Software Research who have extensive experience in the development and marketing of software testing tools. It was first added to STW in 1993, and STW has over 3,500 licenses to more than 1,500 addresses worldwide. METRIC is a language-independent tool supported by translators for Ada, C, C++, and Fortran. The examination was performed on METRIC version 2.10 running on a Sun SPARC Station. At the time of examination, the price for METRIC as a stand-alone product started at $4,000, but when bundled with other STW components this price drops to as low as $775 a copy.

METRIC operates through a graphical user interface that employs pull-down menus. Once invoked, the user starts by specifying the file(s) to be analyzed. This is done by selecting either the Load Single File option and then choosing the appropriate file from the file selection dialog box that pops up, or by selecting the Load Multiple Files option and identifying the set of files to be processed. (The set of files can be identified by a pattern match, by highlighting each file separately, or by specifying the selection of all files shown on the file selection menu.) Once the user specifies that processing should continue, the files(s) are automatically analyzed to determine their complexity.

The analysis generates a Complexity Report and a new window is automatically created to present this report. The types of information included in the report are determined according to a set of predefined default values. Essentially, the Complexity Report lists the encountered program units (in their order of occurrence) and the Software Science and Cyclomatic Complexity measures for each. In order to help determine the most complex program units, METRIC allows the user to specify ordering the presentation of the units in terms of ascending/descending value of any of 17 complexity measures. (Special symbols in front of a program unit name are used to indicate a task or package.)

Additional textual reports are available through one of the menus. A Summary Report presents an accumulated account of the complexity measure for the entire source code analyzed. An Exception Report identifies those program units that violate predefined maximum complexity values. A Generic Report lists, for each encountered generic unit, the generic name, the number of times it is instantiated, and the names of instantiating units. Two special reports are provided for packages. The first of these identifies packages that violate predefined complexity standards. The second provides an intermediate summary report at the package level.

In addition to this textual reporting, METRIC offers kiviat diagrams as a graphical means to view the effect of applying multiple metrics to the source code. Three types of kiviat diagrams are available, each providing an increasing number of metrics to offer more detailed insights into the software. Each of these kiviat diagrams represents information presented in the Summary Report. If desired, users can also define their own diagrams to display selected metric values.

METRIC supports some tailorability through a special menu provided for setting threshold values. Further tailoring is achieved by editing the configuration file; this allows changing, for example, the default page length and the required minimum and maximum estimated lengths of Ada packages.

METRIC also can be operated in command line mode. In this case, the user simply enters the METRIC command optionally followed by a series of options that determine the types of textual output reports that are required. A separate command is provided for producing graphical kiviat diagrams of the generated results.

31.2 Observations

Ease of use. METRIC is very easy to use, requiring no special knowledge on the part of the user. Help boxes are available, although these only offer relatively high-level information on various options. An Error Report is automatically produced if errors are encountered during processing and analysis. This report provides information that is helpful in correcting the error situation.

Documentation and user support. As always, Software Research's documentation is extensive and very useful. In this case, the documentation includes discussions on the development of complexity measures, the use of metrics to support both development and maintenance activities, and a few words on the advisability of tuning metrics based on historical data.

Ada restrictions. METRIC supports the full Ada language.

Problems encountered. No problems were encountered in the use of METRIC.

31.3 Sample Outputs

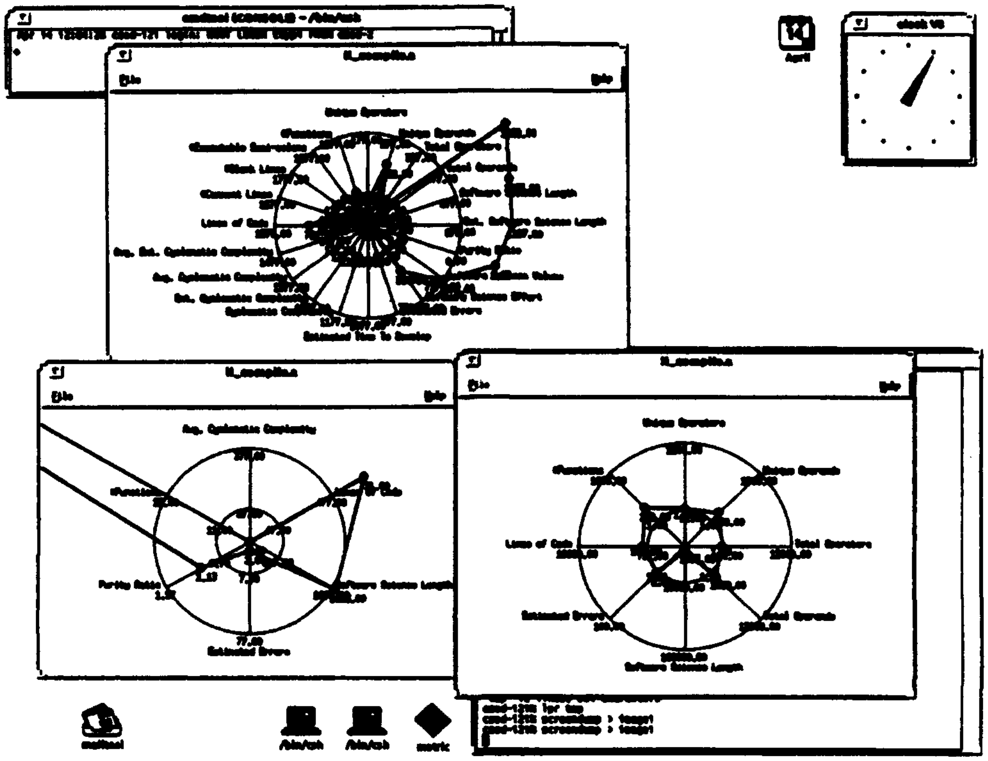
Figures 31-1 through 31-6 provide sample outputs from METRIC on the Ada Lexical Analyzer Generator example.

```
#         #
#         # A Sample of Type-I Kiviat Chart Definition
#         #
#         # Min           Max           Value         Text
1         37            377           4             Avg. Cyclomatic Complexity
1         47            477           5000          Lines of Code
1         107           1007          55000         Software Science Length
1         7             77            50            Estimated Errors
10        0.700         1.970         1.58          Purity Ratio
1         11            27            123           #Functions
10
1
```

```
#
# A Sample of Type-II Kiviat Chart Definition
#
# Min           Max           Value         Text
17            177           100           Unique Operators
27            277           100           Unique Operands
37            377           100           Total Operators
47            477           100           Total Operands
57            577           100           Software Science Length
67            677           100           Est. Software Science Length
0.070         0.700         100           Purity Ratio
77            7777          100           Software Science Volume
87            778877        100           Software Science Effort
97            977           100           Estimated Errors
107           1077          100           Estimated Time To Develop
117           1177          100           Cyclomatic Complexity
127           1277          100           Ext. Cyclomatic Complexity
137           1377          100           Avg. Cyclomatic Complexity
147           1477          100           Avg. Ext. Cyclomatic Complexity
157           1577          100           Lines of Code
167           1677          100           #Comment Lines
177           1777          100           #Blank Lines
187           1877          100           #Executable Semi-colons
197           1977          100           #Functions
```

```
#
# A Sample of Type-II Kiviat Chart Definition
#
# Min           Max           Value         Text
1             1200          10            Unique Operators
1             1000          6             Unique Operands
1             12000         10            Total Operators
1             12000         6             Total Operands
10000         100000        55000         Software Science Length
1             100           50            Estimated Errors
100           10000         5000          Lines of Code
1             1000          123           #Functions
```

Figure 31-1. METRIC Kiviat Chart Definitions



```
cmd-1210 lpr tmp
cmd-1210 screendump > tmpgrf
cmd-1210 screendump > tmpgrf
```

Figure 31-2. METRIC Kiviat Charts for LL_COMPILE

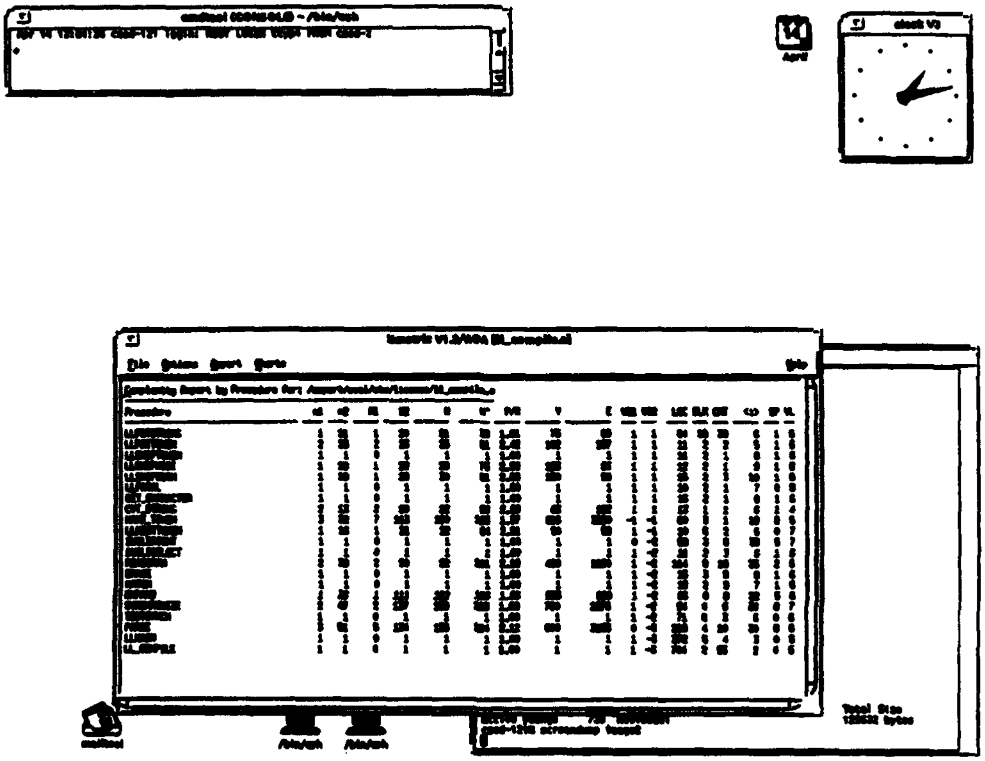


Figure 31-3. METRIC Complexity Report for LL_COMPILE

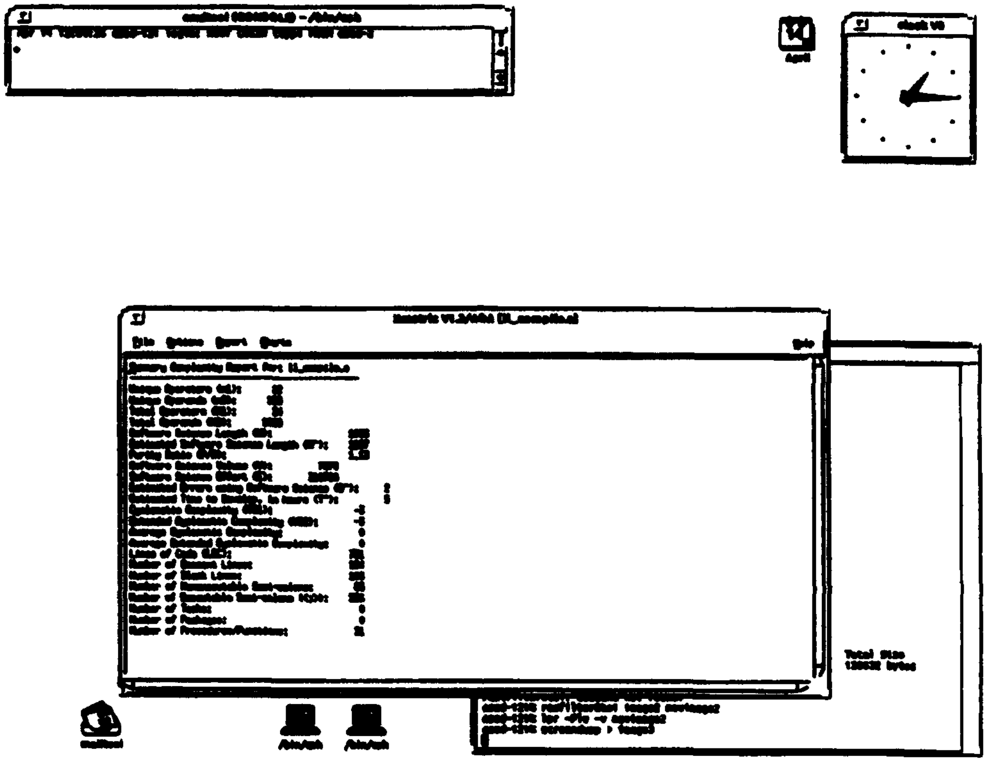


Figure 31-4. METRIC Complexity Summary Report for LL_COMPILE

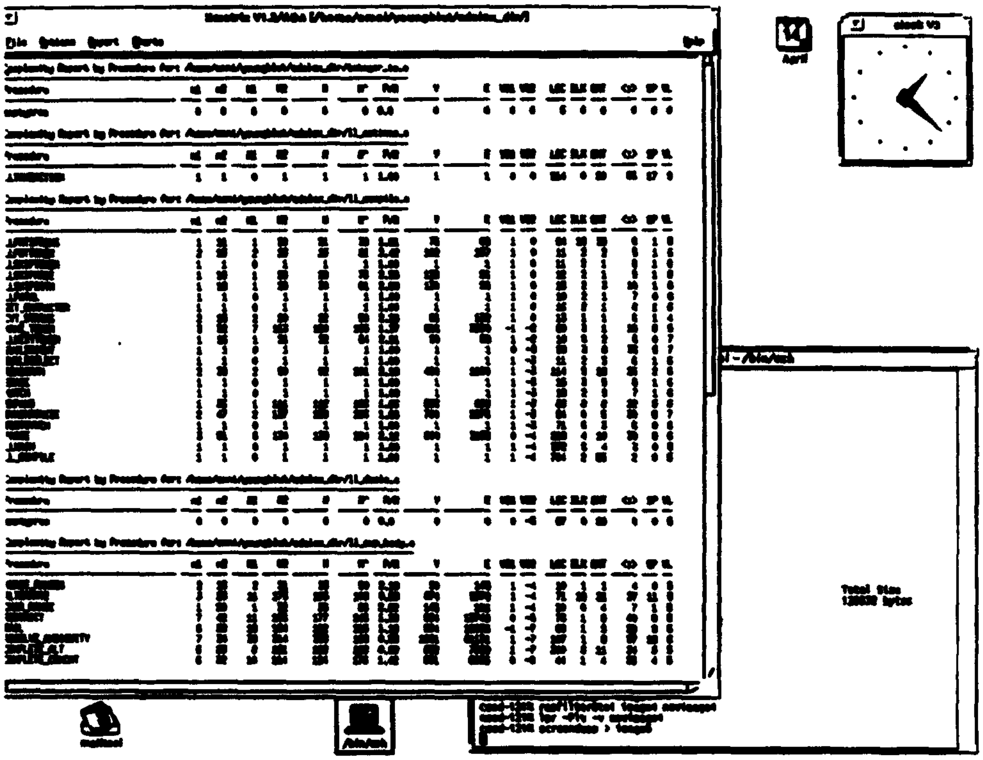
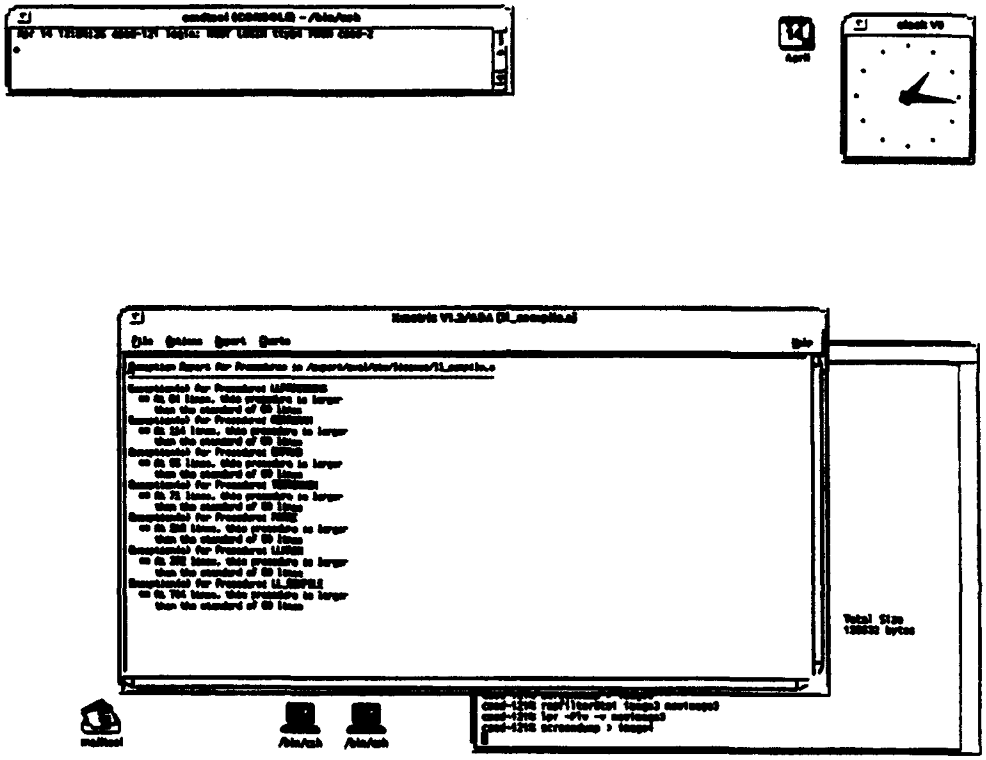


Figure 31-5. METRIC Reporting on Multiple Files



```
Exception Report for Procedures in [illegible]/ll_compile.c

Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure [illegible]
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
Exception(s) for Procedure LL_COMPILE
   => At [illegible] lines, this procedure is larger
      than the standard of [illegible] lines
```

```
Total Size
[illegible] bytes
```

Figure 31-6. METRIC Exception Report for LL_COMPILE

32. McCabe Tools

The Battlemap Analysis Tool (BAT), Analysis of Complexity Tool (ACT), McCabe Instrumentation Tool, SLICE, and CodeBreaker support the McCabe Structured Testing methodology documented in NIST Publication 500-99 [NIST 1982]. BAT supports examining software structure at both the program and module levels, it reports on software complexity, generates call graphs and flowgraphs, and identifies needed test paths. BAT also provides support for object-oriented applications via a Class Editor which allows the user to group modules together into a class and then name it. ACT provides a subset of the BAT functionality focusing on module level analysis. The McCabe Instrumentation Tool supports coverage analysis. SLICE is a data-driven software visualization tool that allows the user to examine the last execution path taken through a program. CodeBreaker compares the structure of modules or programs to aid in the identification of reusable and redundant code, the locating of a particular module within a large system, or identifying the scope of needed regression testing. It also compares program implementation with design.

Additional tools, not examined here, are START, McCabe OO Tool, CaseBridge and BattlePlan. START analyzes Data Flow Diagrams (DFDs) to compute requirements complexity, identifies DFD components which may run asynchronously, and identifies end-to-end-scenarios, both in terms of test conditions necessary to drive each data flow scenario and in terms of data flows for acceptance testing. McCabe OO Tool provide static and dynamic analysis for object-oriented designed software. CaseBridge provides both forward and reverse engineering interfaces between BAT and the StP and Teamwork CASE systems. BattlePlan is a new tool; it is a front-end CASE tool that allows users to design new applications and incorporate new functions into a system within the framework of reverse engineering. The Inference Engine, a BAT option, provides a query language intended to help a user identify where a design needs to be improved; this option was not examined.

McCabe Structured Testing is based on the cyclomatic complexity metric. This metric has been the subject of several evaluations. In 1988, the Air Force Electronic Systems Division (ESD) adopted the measurement and control of cyclomatic complexity on all of its contracts.

32.1 Tool Overview

The McCabe tools were developed by McCabe & Associates. This organization supports a user group and provides a newsletter and hot-line support to tool users. Training seminars and consultancy services are also available. Most of the McCabe tools support over a dozen languages and over two dozen dialects, including Ada, C, C++, Cobol, Pascal, Fortran, and PL/1. The recently introduced McCabe Instrumentation Tool, however, currently only supports Ada, C, C++, Cobol, and Fortran. The tools also support Caine, Faber, and Gordon PDL and those PDLs employed by StP and Teamwork. The tools are available on a wide range of platforms, including Sun, Apollo, HP, and NCR Tower workstations under Unix, DEC VMS and Ultrix systems, and the IBM PC under DOS. They use OSF/ Motif and OpenWindows as appropriate.

The tools are packaged in various ways. ACT, CodeBreaker, and the Instrumentation Tool can be invoked via the graphical McCabe Menu-Driven User Interface. BAT has its own graphical user interface; with the appropriate licenses, CodeBreaker, the Instrumentation Tool, and SLICE can be invoked from the BAT interface.

The evaluation was performed on McCabe tools version V19920601-1 13 CTS running on a Sun SPARCstation under Unix with OpenWindows. At the time of evaluation, the price for ACT started at $11,500 and the price for BAT at $21,500. Other tools are purchased in addition to BAT and the price for CodeBreaker started at $5,000; the price for the McCabe Instrumentation Tool and SLICE, together, started at $5,000. McCabe & Associates will provide prospective users with a working example of the users' own program. Additionally, demonstration discs for ACT are currently available, and expected to become available for the McCabe Instrumentation Tool in the near future.

Some aspects of tool usage are common to all of the McCabe tools. All Ada source programs must be parsed before the tools can be run. Several different parsers are available for Ada source code, the choice of which to use depends on the task at hand, for example instrumentation or analyzing Halstead metrics. Each parser translates Ada source code into McCabe intermediate files. By default, the parse library resides in the current directory; the user must be careful when moving between directories not to create multiple parse libraries and also to ensure that successive parses do not corrupt the parse library. While parsing in correct compilation order is not necessary to use the full capabilities of ACT, it is recommended. Access to the full capabilities of BAT requires parsing in compilation order.

The tools use two configuration files to specify needed information. The System Configuration File defines, for example, the chart plotter driver and applicable parsers. The Program Configuration File defines the source files that make up each program.

Finally, tool operation can be modified via a set of commands given in an *exclude file*. (Although the use of exclude files is only available under a BAT license, these special files can be used to affect the output of other tools as well.) Exclude files can be used to limit the level of calls from a module, exclude modules from analysis, or bind together classes of similar modules. They can be used via either the graphical or command line interfaces, or through special entries in the Program Configuration File.

32.1.1 ACT

The ACT has two primary purposes: examining the control structure of a module and generating the set of test paths needed for basis path testing. Basis path testing is a form of structured testing where the minimum set of linearly independent paths through a module are exercised. ACT can be applied to both PDL and source code.

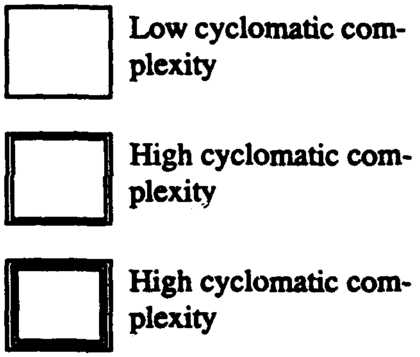
Using one of the graphical user interfaces, the user starts the application of ACT by identifying the program, file(s), or module(s) of interest. The various options and tool functions are then invoked via pull-down menus. To aid in the examination of module control structure, ACT generates flowgraphs that give a graphical overview of the structure of each module. This special type of directed graph uses different colors, or line types if color is not available, to distinguish between upward flows, a structured exit from a loop, and other plain downward flowing edges. Similarly, edges for loops or loop exits are shown as curved lines, whereas most other edges are straight lines. Flowgraphs are supported by annotated source listings that relate a graph back to the source code. The final capability provided in this category is the analysis of module complexity. ACT reports on complexity in terms of McCabe's cyclomatic, essential, and module design complexity, and using lines of code and Halstead's metrics.

ACT supports McCabe Structured Testing by deriving a basis set of end-to-end test paths for a module. These test paths can be presented graphically as a path through the flowgraph (either superimposed on the flowgraph or showing test paths only) or listed textually with their corresponding node numbers and test conditions. Some of the generated paths may be unrealizable, for example, where there are data dependencies between conditions that prevent a condition from taking on the values required to execute its branches. ACT also gives the user the option to define his own set of test paths. This capability allows, for example, eliminating unrealizable paths from the set of test paths derived.

32.1.2 BAT

BAT extends the functionality of ACT is several ways; primarily, it extends the static analysis to address the integration complexity of source code and the generation of integration test paths. The following discussion just addresses the additional functionality provided by BAT. Like ACT, BAT can be applied to PDL in addition to source code.

On its invocation, BAT starts by presenting a Battlemap display of the program under examination. The main part of a Battlemap display is a structure chart that graphically displays the modules in a program, or subsystem, and the calling relationships between these modules. The cyclomatic and essential complexity of each module can be shown on a Battlemap using the following symbols:


Low cyclomatic complexity
High cyclomatic complexity
High cyclomatic complexity

Alternatively, on a color monitor, colors can be used to represent different complexity levels. Special notations to represent iterative and conditional calls between modules are available. Structure charts can be drawn in footprint, terse, or verbose detail, essentially affecting how modules are identified. View options allow adjusting structure chart layout and positioning within the structure chart; in this last case, the user can search for a particular module on a large Battlemap display and the display will automatically reposition with respect to this module. A variety of information can be called up by clicking on a module. This includes the annotated source code listing, module flowgraph, module summary, test path listings, and test path graphs. Free-form messages or specifications pertaining to a selected module and even, on a Sun SPARCstation, audio notes to annotate a selected module can be entered and retrieved. By clicking on a line between modules, the user can request a line summary report. This gives the numbers of the connected modules, line type (unconditional, conditional, iterative), and any invoked parameters. A Battlemap display can be printed as a whole or as a set of subtrees. The print of the entire structure chart can be compressed onto a single page or spread across several pages with special switches controlling the use of off-page connector symbols. Other switches allow the user to set the detail level and request the display of formal parameters on the printed structure chart.

Under its graphical interface, all BAT tools and options are invoked from the Battlemap display using pull-down menus. In addition to the cyclomatic and design complexity analysis performed by ACT, BAT also examines essential complexity at the module level to give a feel for the degree to which a module contains unstructured constructs. Cross-referencing between modules is given by listings of called-by and calls-to relationships. Additional textual information supporting the Battlemap display is a context listing showing where modules are located and an index listing that maps module numbers to module names. These listings can be sorted alphabetically, numerically, by position on Battlemap, or by complexity value.

To aid in understanding the design structure of a program, the user can request BAT to generate the design subtrees embodied in the program. These subtrees are superimposed on the Battlemap display and the user can step forward and backward through successive ones. Textual listings of subtrees can also be generated, in this case giving the associated end-to-end test conditions for each subtree that will support structured testing at the integration level. Subtree graphs and text can be generated at both the design and cyclomatic complexity detail levels.

BAT performs static analysis on the program structure to report on its design complexity and integration complexity. Both these metrics are based on module design complexity and take account of the program's architecture. Branch count metrics report on the number of branches contained in each module. In addition to textual listings, graphical representations of the McCabe and Halstead metrics are available. These graphical representations are a histogram, scatterplot, or kiviat diagram where the user can portray selected default metrics or choose from a menu of the full set of McCabe and Halstead metrics. The user can also change the threshold values assigned to chosen metrics.

32.1.3 McCabe Instrumentation Tool

The McCabe Instrumentation Tool works with either ACT, for structural coverage analysis at the unit level, or BAT, to support coverage analysis at both unit and integration levels. Instrumentation is simple and performed similarly under both tools. When performing instrumentation from BAT, for example, the user only needs to specify the destination of the instrumented files (necessary since otherwise original source files will be overwritten), the files to be instrumented, and the instrumenting parser in the Program Configuration File, and request the *export* option. The instrumented files are then generated and written to the specified directory. After the provided McCabe library of instrumentation routines has been compiled, the instrumented files are compiled and linked as usual. When run, the instrumented program creates a file of trace data which is subsequently *imported* back to the BAT environment for analysis. BAT maintains a data base of test results so the results of successive test runs can be accumulated for cumulative coverage reporting.

Coverage information is available textually and, using the capabilities of either ACT or BAT as appropriate, graphically to provide a visual representation of test effectiveness. At the module level, the tool reports on the tested, untested, and partially tested basis paths. Tested and untested paths can be listed textually or graphically by, for example, superimposing paths on the module flowgraph. A branch coverage report shows the number of branches, the number tested, and the percent of branches tested for each module. In this case the graphical equivalent is an edge coverage graph which is a subset of the full module flowgraph (edges that do not lie on a tested path are omitted) and which gives a visual overview of unit test coverage. Finally, the actual complexity is compared to cyclomatic complexity to give a module testedness report that can be used to identify modules requiring additional attention.

Similar types of information are provided at the integration level, but here reports address tested, untested, and partially tested subtrees. Again, information is available both textually and graphically (this time superimposed on the Battlemap display). Two new metrics are used to relate achieved coverage to desired coverage. The design testedness metric provides an overview of the testedness of the entire program, whereas the integration complexity metric summarizes the number of subtrees at the design detail level.

32.1.4 SLICE

A software slice is defined as the set of code touched by one execution trace through a program. Visualization of a slice is useful in, for example, debugging to identify where a bug is located and the path and code associated with the bug. It can also be used to aid in establishing the traceability between requirements and physical code by picking test data that is representative of a particular functional requirements and obtaining the corresponding execution slice.

The SLICE tool determines slices dynamically as a program executes with chosen test data. It requires both BAT and the McCabe Instrumentation Tool. First the program is instrumented to produce a trace file, executed with sample data and the trace file imported back into BAT as discussed previously. The user then can access SLICE data in one of two ways. To request SLICE information for a single module, the user can click on that module on the Battlemap. BAT then displays the slice of code that was traversed during that module's execution. The slice is shown both graphically and textually. For the graphical view, the module's edge coverage graph is shown with the edges that were traversed during execution being denoted by a solid line. The textual view is presented alongside with the executed path through the associated source code highlighted.

To view slice information for the program as a whole, or for several modules, the user switches to the instrumentation mode within BAT; this causes the SLICE options dialog box to be displayed. Here, again, information is presented both graphically and textually. In this case, however, the edge coverage graph and associated text is presented for each module in turn, in the order that the modules were executed. The user can step forward or backward through the set of traversed modules. Other SLICE options allow the highlighted text and edge coverage graph to be printed.

32.1.5 CodeBreaker

CodeBreaker is primarily used to compare the structure of two modules or two programs. This is used for purposes such as verifying that the restructured version of a module preserves both the decision structure and the calling structure of the original, comparing an original and modified module to determine the design paths that need regression testing, and aiding in the identification of likely candidates for further examination as possible redundant or reusable modules.

The user can request the comparison of modules contained in either two specified files or two specified programs. CodeBreaker then compares the designs of the selected modules. It does this by generating a basis set of expanded or design-reduced paths through the first module and examining each corresponding path through the second module for matches or mismatches. The user can set switches that control what the tool classes as a mismatch; these switches cause CodeBreaker to ignore condition names, module names, or module calls. An additional switch specifies that attempts should be made to match the secondary module to any subgraph of the primary module. To aid in identifying likely candidates for comparison, the user can also request a list of the modules in two programs sorted by the weighted sums of their complexity metrics.

Program comparison is performed in a similar manner. Here CodeBreaker generates a basis set of integration paths through the first program and attempts to find a corresponding subtree through the second program that "matches" each of these paths. Again the user can request that module calls, modules names, and condition names be ignored. In this case, an additional switch allows the user to specify the required level of comparison in terms of module invocation depth.

For both module and program comparison, the output depends on the choice of which object is treated as the primary object and which the secondary. For complete comparison results, each object should be examined in both roles.

32.2 Observations

Ease of use. Most of the tools have a graphical menu-driven interface for use with windows and a command-driven interface for use when windowing is not available. In all cases, the interfaces offer consistent usage; for example, if no program name is given for a tool invocation, the default is taken from the Program Configuration File. The graphical interfaces are uniformly easy to use. While not actually difficult to use, the command line interface can require the user to give many arguments to invoke full tool flexibility and power. This interface helps the user by allowing him to request information on the arguments available with each command type.

Minor tailorability is allowed through the use of the `system.def` file. This file is used to specify editor, parser, printer, and plotter information. It also provides for customization of font format, graph layout and notation, use of colors, threshold values for various metrics, and special options for some tools. In addition, special environment variables allow, for example, specifying a directory for the parse library to reside in. Users can have their own version of the `system.def` file.

In addition to supporting flow drivers for HPGL plotters, HP Laserjet compatible output for Unix and DOS, Postscript output, and *scrn* for DOS, these tools also support a zoomflow driver that generates output for X-Windows and OpenLook users, allowing the user to enlarge or reduce flowgraphs presented on the screen.

Documentation and user support. The tools are supported with extensive documentation. In a couple of cases, the documentation did not exactly match tool operation, but these slight deviations caused little difficultly. McCabe & Associates was very helpful and prompt in answering any questions that arose.

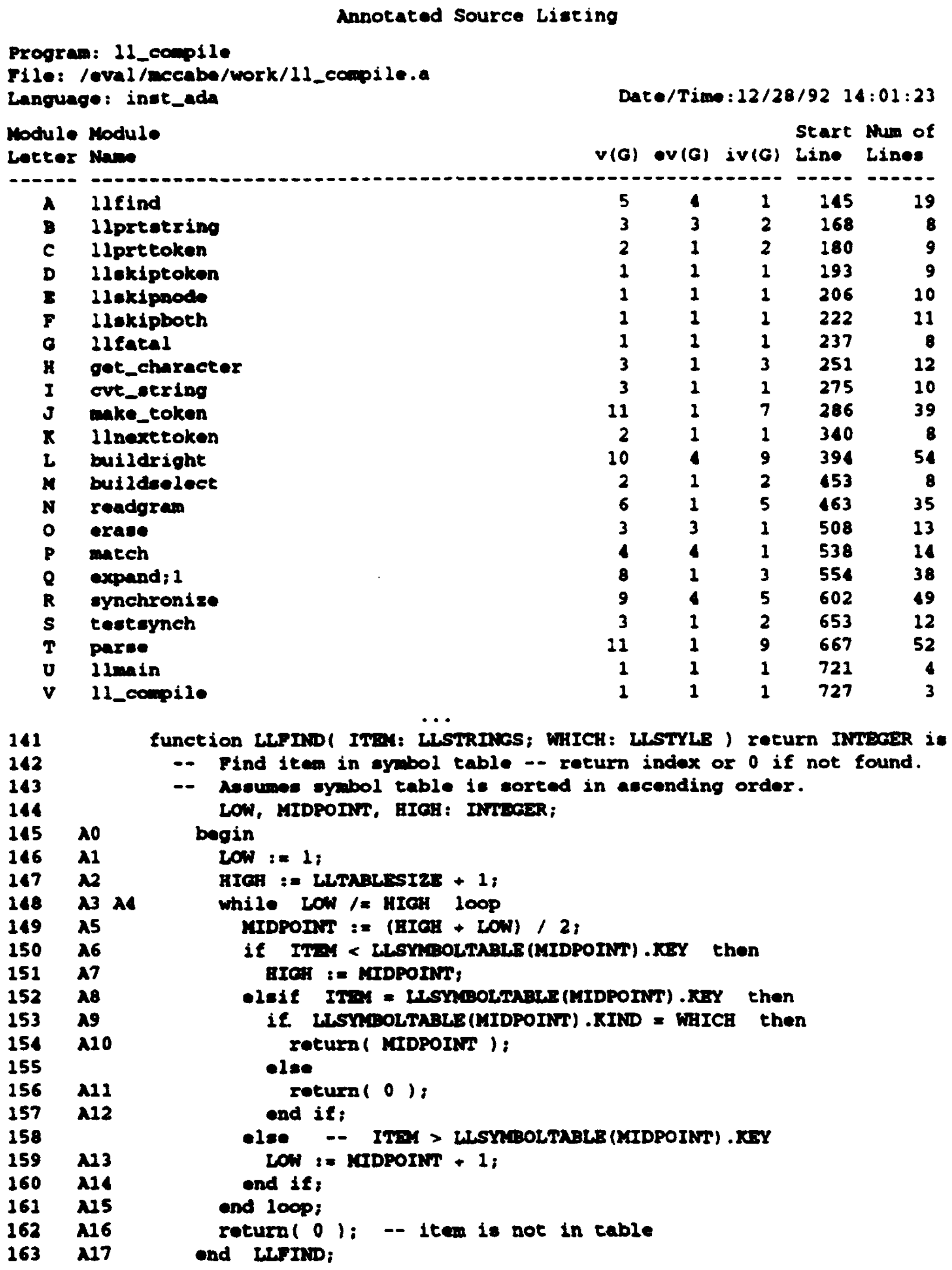
Instrumentation Overhead. The user can limit the amount of instrumentation performed by explicitly identifying the files to instrument. Additionally, using the command line interface, the user can specify which modules within a file to instrument. Full instrumentation of the Ada Lexical Analyzer Generator gave an increase in source code size of 28% and an increase in object code size of 15%.

Ada restrictions. These tools do not handle concurrent Ada programs. Tasks are treated as sequential subprograms. Specifically, the *select* statement is treated similar to a *case* statement, *accept* and *abort* statements are ignored, a *terminate* alternative is treated as a *return* statement, and a remote procedure call referring to an entry declared in another task (*t*) is translated to a call to the task (*t*) itself. The Ada parser ignores exceptions that can be raised implicitly by means of exception propagation. An explicit *raise* statement is always translated to a *return* statement regardless of whether a handler is associated with the exception. For convenience, all handlers attached to the end of a frame are grouped together like a *case* statement and treated as a separate module. Extended names are not used and the parser differentiates between overloaded or redeclared module names by assigning version numbers to module names in the order of their arrival to the parser. Since the parser does not remember the types and object declarations in programs, it is not able to perform overload resolution when parsing a subprogram invocation that referred to an overloaded subprogram name. Instead, it chooses arbitrarily from among the module names that correspond to the overloaded subprograms. This may result in inaccuracies in the BAT outputs.

Problems encountered. No problems were encountered in tool operation.

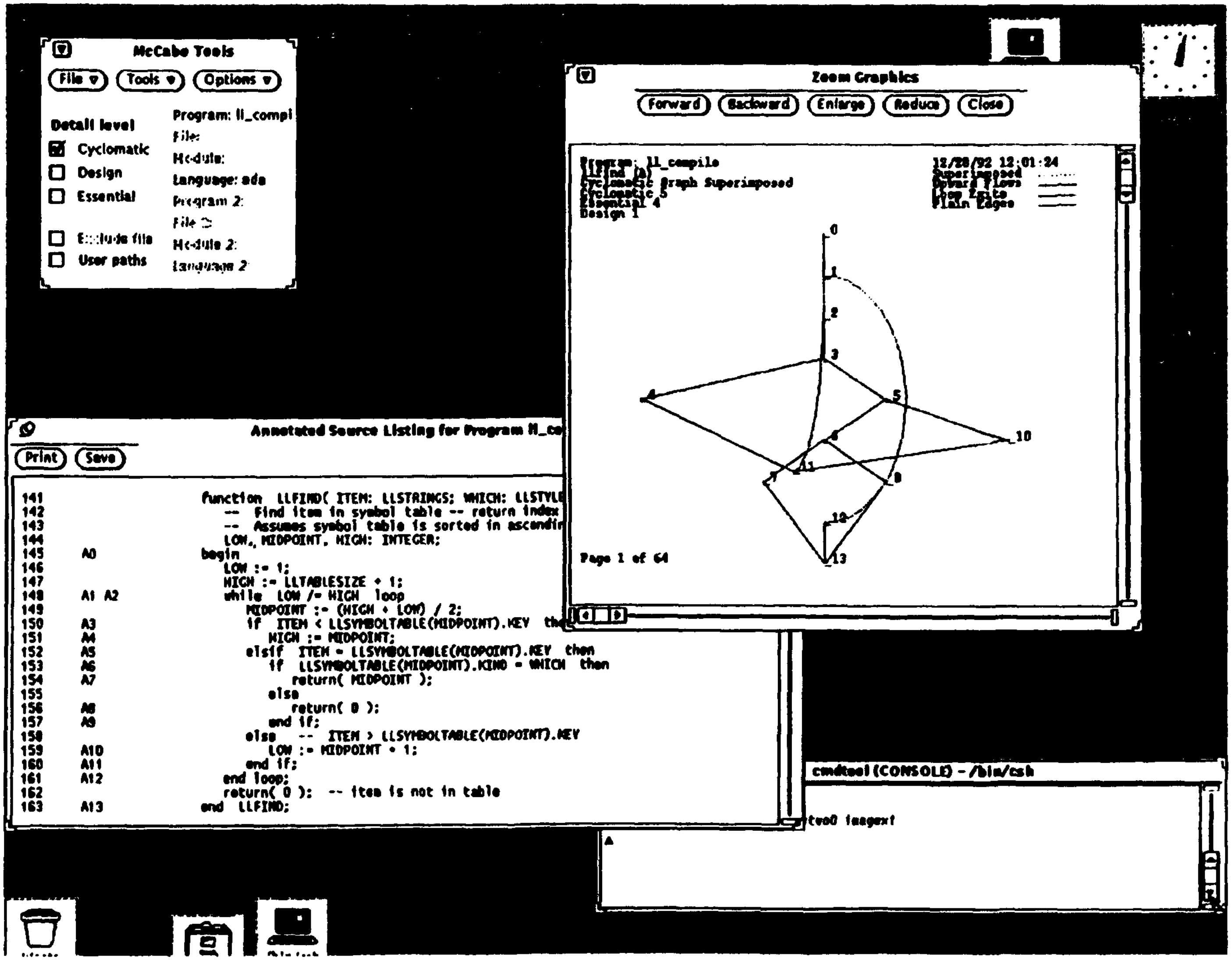
32.3 Sample Outputs

Figures 32-1 through 32-40 provide sample outputs from McCabe tools.



```
                         Annotated Source Listing

Program: ll_compile
File: /eval/mccabe/work/ll_compile.a
Language: inst_ada                               Date/Time:12/28/92 14:01:23

Module Module                                                   Start  Num of
Letter Name                                 v(G)  ev(G)  iv(G)  Line   Lines
------ ------------------------------------------------------- ----- ------
   A   llfind                                  5      4      1   145     19
   B   llprtstring                             3      3      2   168      8
   C   llprttoken                              2      1      2   180      9
   D   llskiptoken                             1      1      1   193      9
   E   llskipnode                              1      1      1   206     10
   F   llskipboth                              1      1      1   222     11
   G   llfatal                                 1      1      1   237      8
   H   get_character                           3      1      3   251     12
   I   cvt_string                              3      1      1   275     10
   J   make_token                             11      1      7   286     39
   K   llnexttoken                             2      1      1   340      8
   L   buildright                             10      4      9   394     54
   M   buildselect                             2      1      2   453      8
   N   readgram                                6      1      5   463     35
   O   erase                                   3      3      1   508     13
   P   match                                   4      4      1   538     14
   Q   expand;1                                8      1      3   554     38
   R   synchronize                             9      4      5   602     49
   S   testsynch                               3      1      2   653     12
   T   parse                                  11      1      9   667     52
   U   llmain                                  1      1      1   721      4
   V   ll_compile                              1      1      1   727      3
                                        ...
141          function LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE ) return INTEGER is
142            -- Find item in symbol table -- return index or 0 if not found.
143            -- Assumes symbol table is sorted in ascending order.
144              LOW, MIDPOINT, HIGH: INTEGER;
145    A0      begin
146    A1        LOW := 1;
147    A2        HIGH := LLTABLESIZE + 1;
148    A3 A4     while  LOW /= HIGH  loop
149    A5          MIDPOINT := (HIGH + LOW) / 2;
150    A6          if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
151    A7            HIGH := MIDPOINT;
152    A8          elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
153    A9            if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
154    A10             return( MIDPOINT );
155                  else
156    A11             return( 0 );
157    A12           end if;
158                else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
159    A13           LOW := MIDPOINT + 1;
160    A14         end if;
161    A15       end loop;
162    A16       return( 0 );  -- item is not in table
163    A17     end  LLFIND;
```

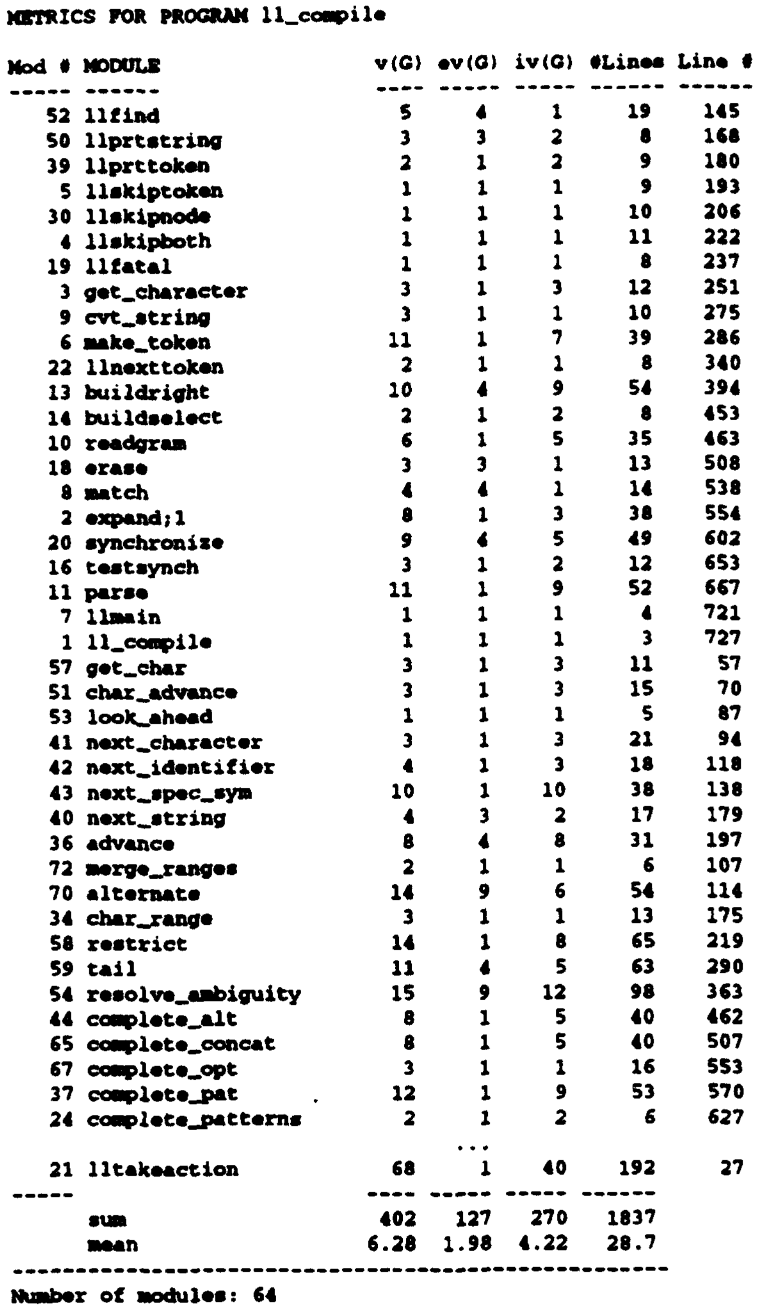
Figure 32-1. ACT Annotated Listing for Function LLFIND



```
Program: ll_compile
llfind (A)
Cyclomatic Graph Superimposed
Cyclomatic 5
Essential 4
Design 1
12/28/92 12:01:24
```

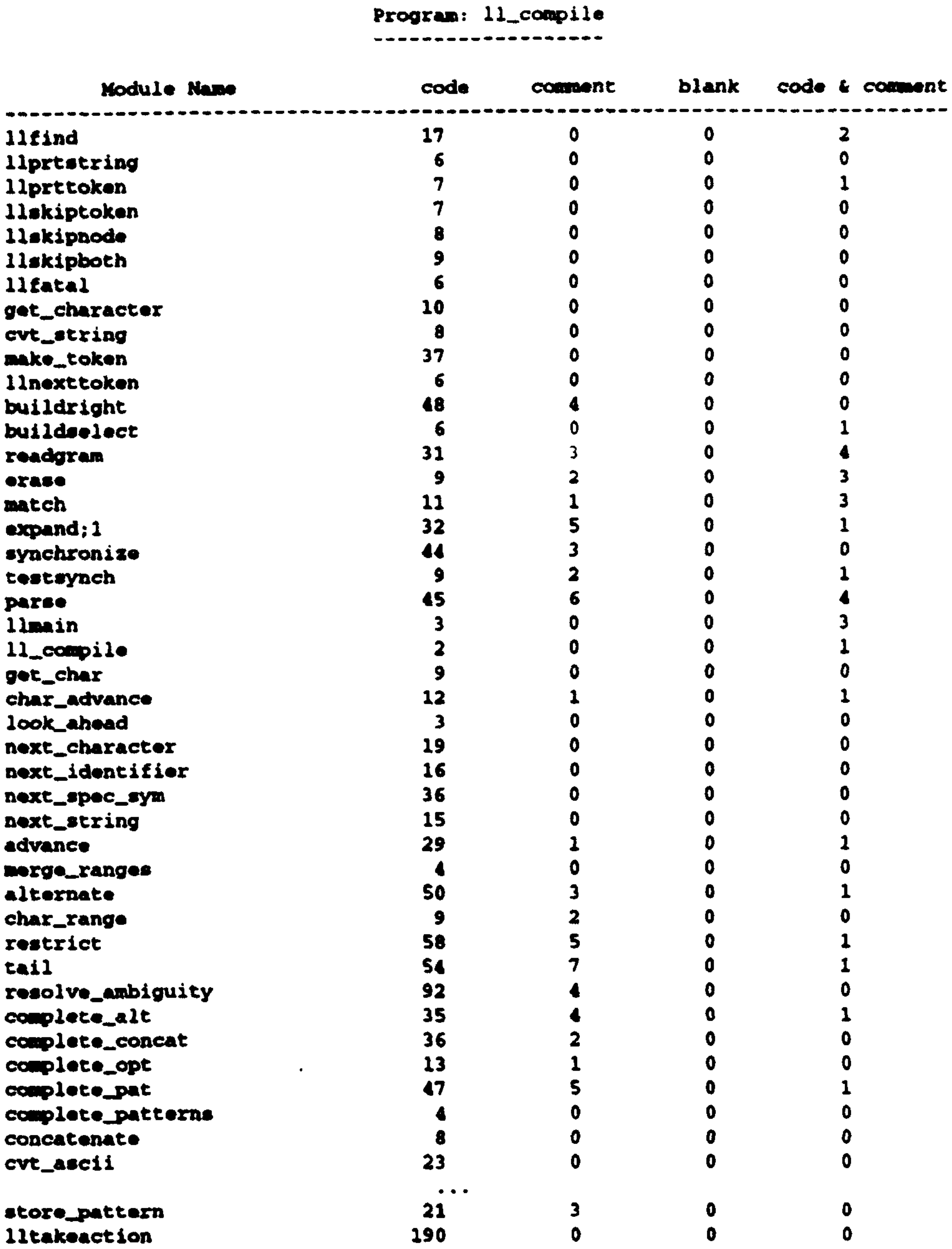
```
                function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE
                   --  Find item in symbol table -- return index
                   --  Assumes symbol table is sorted in ascendin
                   LOW., MIDPOINT, HIGH: INTEGER;
        A0      begin
                   LOW := 1;
                   HIGH := LLTABLESIZE + 1;
        A1 A2      while  LOW /= HIGH  loop
                      MIDPOINT := (HIGH + LOW) / 2;
        A3            if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  the
        A4               HIGH := MIDPOINT;
        A5            elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
        A6               if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
        A7                  return( MIDPOINT );
                         else
        A8                  return( 0 );
        A9               end if;
                      else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
        A10              LOW := MIDPOINT + 1;
        A11           end if;
        A12        end loop;
                   return( 0 );  -- item is not in table
        A13     end  LLFIND;
```

Figure 32-2. ACT Graph for LLFIND



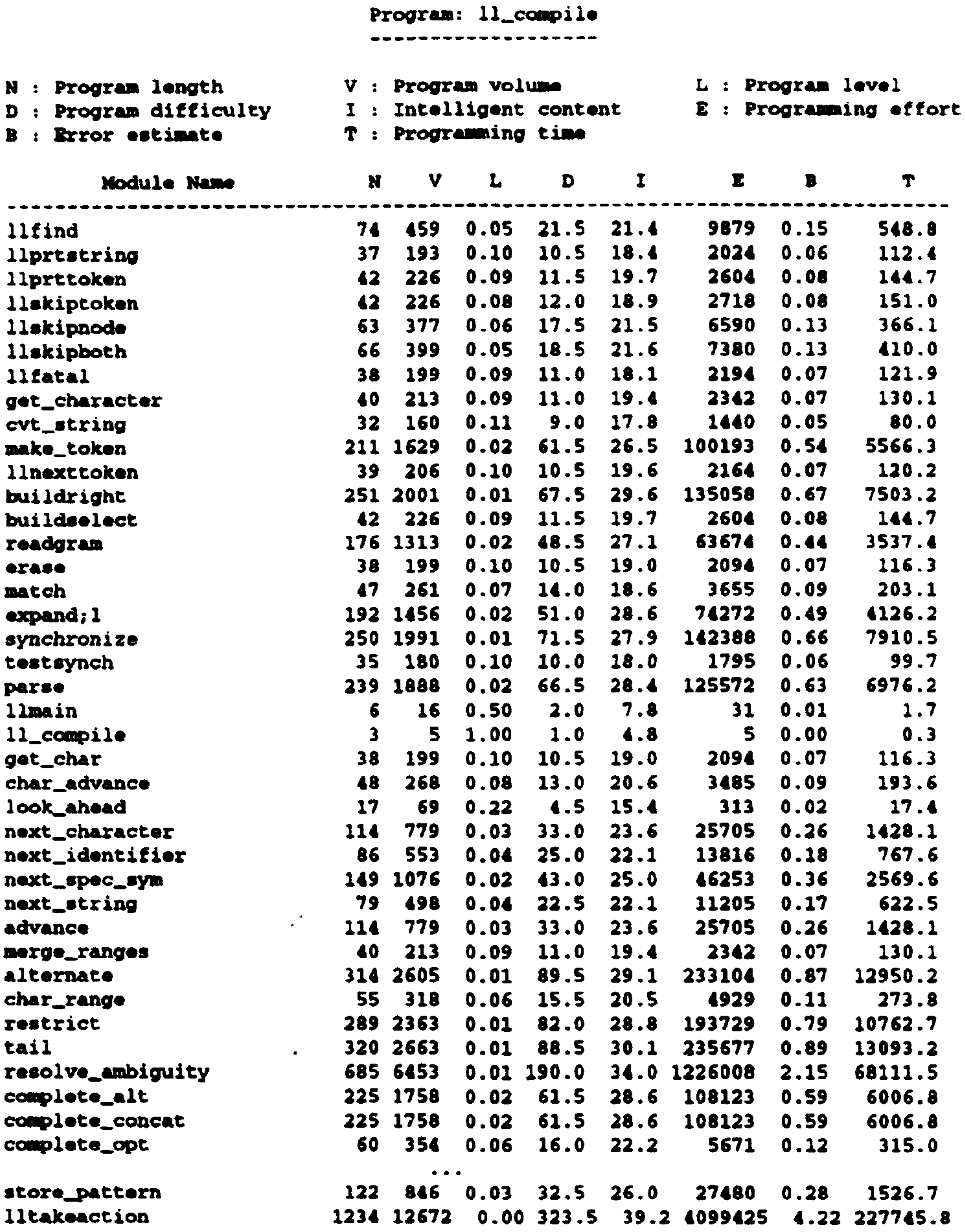
```
METRICS FOR PROGRAM ll_compile

Mod # MODULE                  v(G) ev(G) iv(G) #Lines Line #
----- ------                  ---- ----- ----- ------ ------
   52 llfind                     5     4     1     19    145
   50 llprtstring                3     3     2      8    168
   39 llprttoken                 2     1     2      9    180
    5 llskiptoken                1     1     1      9    193
   30 llskipnode                 1     1     1     10    206
    4 llskipboth                 1     1     1     11    222
   19 llfatal                    1     1     1      8    237
    3 get_character              3     1     3     12    251
    9 cvt_string                 3     1     1     10    275
    6 make_token                11     1     7     39    286
   22 llnexttoken                2     1     1      8    340
   13 buildright                10     4     9     54    394
   14 buildselect                2     1     2      8    453
   10 readgram                   6     1     5     35    463
   18 erase                      3     3     1     13    508
    8 match                      4     4     1     14    538
    2 expand;1                   8     1     3     38    554
   20 synchronize                9     4     5     49    602
   16 testsynch                  3     1     2     12    653
   11 parse                     11     1     9     52    667
    7 llmain                     1     1     1      4    721
    1 ll_compile                 1     1     1      3    727
   57 get_char                   3     1     3     11     57
   51 char_advance               3     1     3     15     70
   53 look_ahead                 1     1     1      5     87
   41 next_character             3     1     3     21     94
   42 next_identifier            4     1     3     18    118
   43 next_spec_sym             10     1    10     38    138
   40 next_string                4     3     2     17    179
   36 advance                    8     4     8     31    197
   72 merge_ranges               2     1     1      6    107
   70 alternate                 14     9     6     54    114
   34 char_range                 3     1     1     13    175
   58 restrict                  14     1     8     65    219
   59 tail                      11     4     5     63    290
   54 resolve_ambiguity         15     9    12     98    363
   44 complete_alt               8     1     5     40    462
   65 complete_concat            8     1     5     40    507
   67 complete_opt               3     1     1     16    553
   37 complete_pat              12     1     9     53    570
   24 complete_patterns          2     1     2      6    627
                                     ...
   21 lltakeaction              68     1    40    192     27
-----                         ---- ----- ----- ------
      sum                      402   127   270   1837
      mean                    6.28  1.98  4.22   28.7
-------------------------------------------------------------
Number of modules: 64
```

Figure 32-3. ACT Metrics Summary



```
                              Program: ll_compile
                              -------------------

       Module Name                    code     comment     blank   code & comment
-----------------------------------------------------------------------------
llfind                                 17         0          0          2
llprtstring                             6         0          0          0
llprttoken                              7         0          0          1
llskiptoken                             7         0          0          0
llskipnode                              8         0          0          0
llskipboth                              9         0          0          0
llfatal                                 6         0          0          0
get_character                          10         0          0          0
cvt_string                              8         0          0          0
make_token                             37         0          0          0
llnexttoken                             6         0          0          0
buildright                             48         4          0          0
buildselect                             6         0          0          1
readgram                               31         3          0          4
erase                                   9         2          0          3
match                                  11         1          0          3
expand;1                               32         5          0          1
synchronize                            44         3          0          0
testsynch                               9         2          0          1
parse                                  45         6          0          4
llmain                                  3         0          0          3
ll_compile                              2         0          0          1
get_char                                9         0          0          0
char_advance                           12         1          0          1
look_ahead                              3         0          0          0
next_character                         19         0          0          0
next_identifier                        16         0          0          0
next_spec_sym                          36         0          0          0
next_string                            15         0          0          0
advance                                29         1          0          1
merge_ranges                            4         0          0          0
alternate                              50         3          0          1
char_range                              9         2          0          0
restrict                               58         5          0          1
tail                                   54         7          0          1
resolve_ambiguity                      92         4          0          0
complete_alt                           35         4          0          1
complete_concat                        36         2          0          0
complete_opt                           13         1          0          0
complete_pat                           47         5          0          1
complete_patterns                       4         0          0          0
concatenate                             8         0          0          0
cvt_ascii                              23         0          0          0
                                       ...
store_pattern                          21         3          0          0
lltakeaction                          190         0          0          0
```

Figure 32-4. ACT Line Count Report

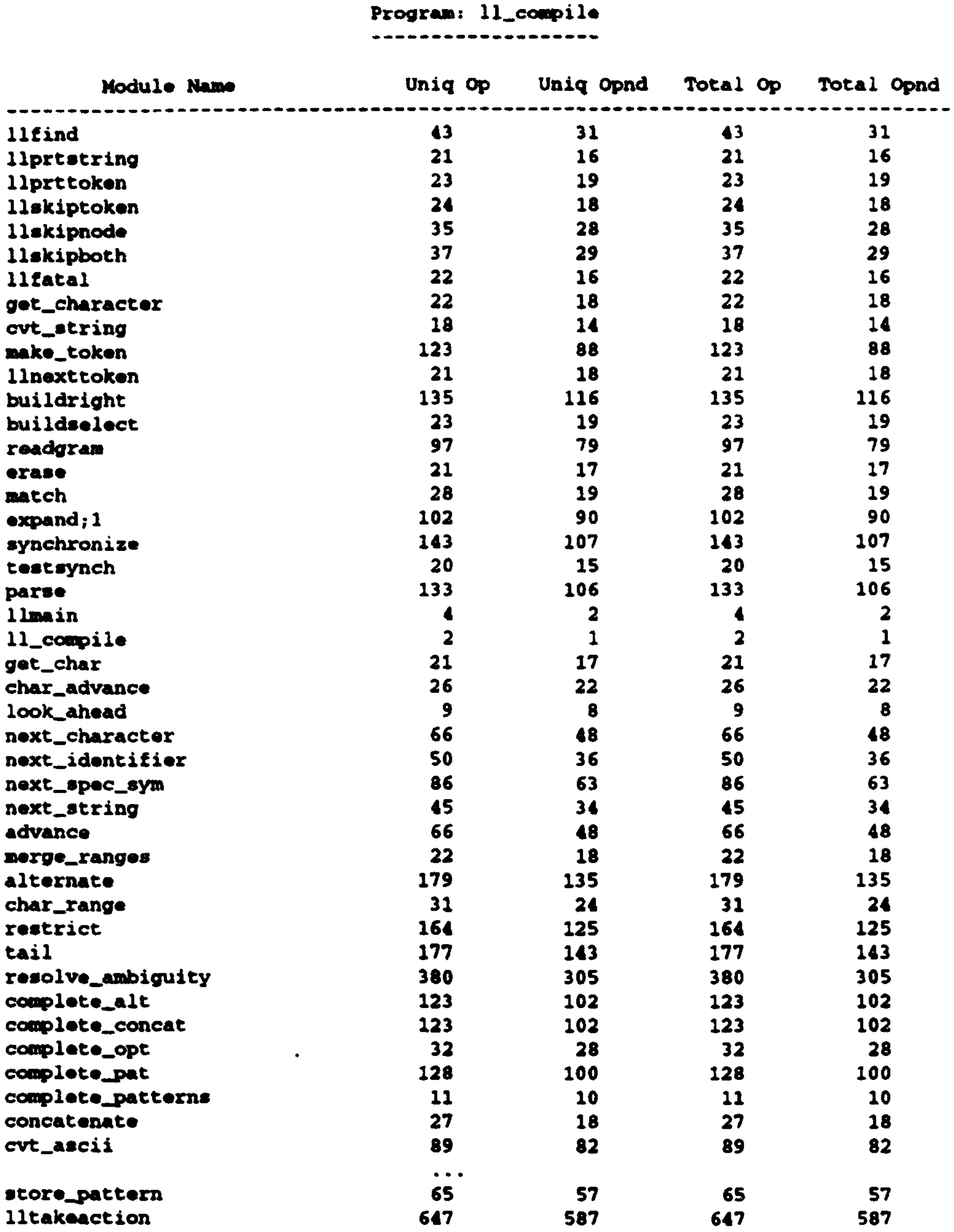


```
                          Program: ll_compile
                          -------------------

N : Program length           V : Program volume           L : Program level
D : Program difficulty       I : Intelligent content      E : Programming effort
B : Error estimate           T : Programming time

        Module Name             N     V     L      D      I        E      B         T
-----------------------------------------------------------------------------------------
llfind                         74   459  0.05   21.5   21.4     9879   0.15     548.8
llprtstring                    37   193  0.10   10.5   18.4     2024   0.06     112.4
llprttoken                     42   226  0.09   11.5   19.7     2604   0.08     144.7
llskiptoken                    42   226  0.08   12.0   18.9     2718   0.08     151.0
llskipnode                     63   377  0.06   17.5   21.5     6590   0.13     366.1
llskipboth                     66   399  0.05   18.5   21.6     7380   0.13     410.0
llfatal                        38   199  0.09   11.0   18.1     2194   0.07     121.9
get_character                  40   213  0.09   11.0   19.4     2342   0.07     130.1
cvt_string                     32   160  0.11    9.0   17.8     1440   0.05      80.0
make_token                    211  1629  0.02   61.5   26.5   100193   0.54    5566.3
llnexttoken                    39   206  0.10   10.5   19.6     2164   0.07     120.2
buildright                    251  2001  0.01   67.5   29.6   135058   0.67    7503.2
buildselect                    42   226  0.09   11.5   19.7     2604   0.08     144.7
readgram                      176  1313  0.02   48.5   27.1    63674   0.44    3537.4
erase                          38   199  0.10   10.5   19.0     2094   0.07     116.3
match                          47   261  0.07   14.0   18.6     3655   0.09     203.1
expand;1                      192  1456  0.02   51.0   28.6    74272   0.49    4126.2
synchronize                   250  1991  0.01   71.5   27.9   142388   0.66    7910.5
testsynch                      35   180  0.10   10.0   18.0     1795   0.06      99.7
parse                         239  1888  0.02   66.5   28.4   125572   0.63    6976.2
llmain                          6    16  0.50    2.0    7.8       31   0.01       1.7
ll_compile                      3     5  1.00    1.0    4.8        5   0.00       0.3
get_char                       38   199  0.10   10.5   19.0     2094   0.07     116.3
char_advance                   48   268  0.08   13.0   20.6     3485   0.09     193.6
look_ahead                     17    69  0.22    4.5   15.4      313   0.02      17.4
next_character                114   779  0.03   33.0   23.6    25705   0.26    1428.1
next_identifier                86   553  0.04   25.0   22.1    13816   0.18     767.6
next_spec_sym                 149  1076  0.02   43.0   25.0    46253   0.36    2569.6
next_string                    79   498  0.04   22.5   22.1    11205   0.17     622.5
advance                       114   779  0.03   33.0   23.6    25705   0.26    1428.1
merge_ranges                   40   213  0.09   11.0   19.4     2342   0.07     130.1
alternate                     314  2605  0.01   89.5   29.1   233104   0.87   12950.2
char_range                     55   318  0.06   15.5   20.5     4929   0.11     273.8
restrict                      289  2363  0.01   82.0   28.8   193729   0.79   10762.7
tail                          320  2663  0.01   88.5   30.1   235677   0.89   13093.2
resolve_ambiguity             685  6453  0.01  190.0   34.0  1226008   2.15   68111.5
complete_alt                  225  1758  0.02   61.5   28.6   108123   0.59    6006.8
complete_concat               225  1758  0.02   61.5   28.6   108123   0.59    6006.8
complete_opt                   60   354  0.06   16.0   22.2     5671   0.12     315.0
                                    ...
store_pattern                 122   846  0.03   32.5   26.0    27480   0.28    1526.7
lltakeaction                 1234 12672  0.00  323.5   39.2  4099425   4.22  227745.8
```

Figure 32-5. ACT Halstead's Metrics Report

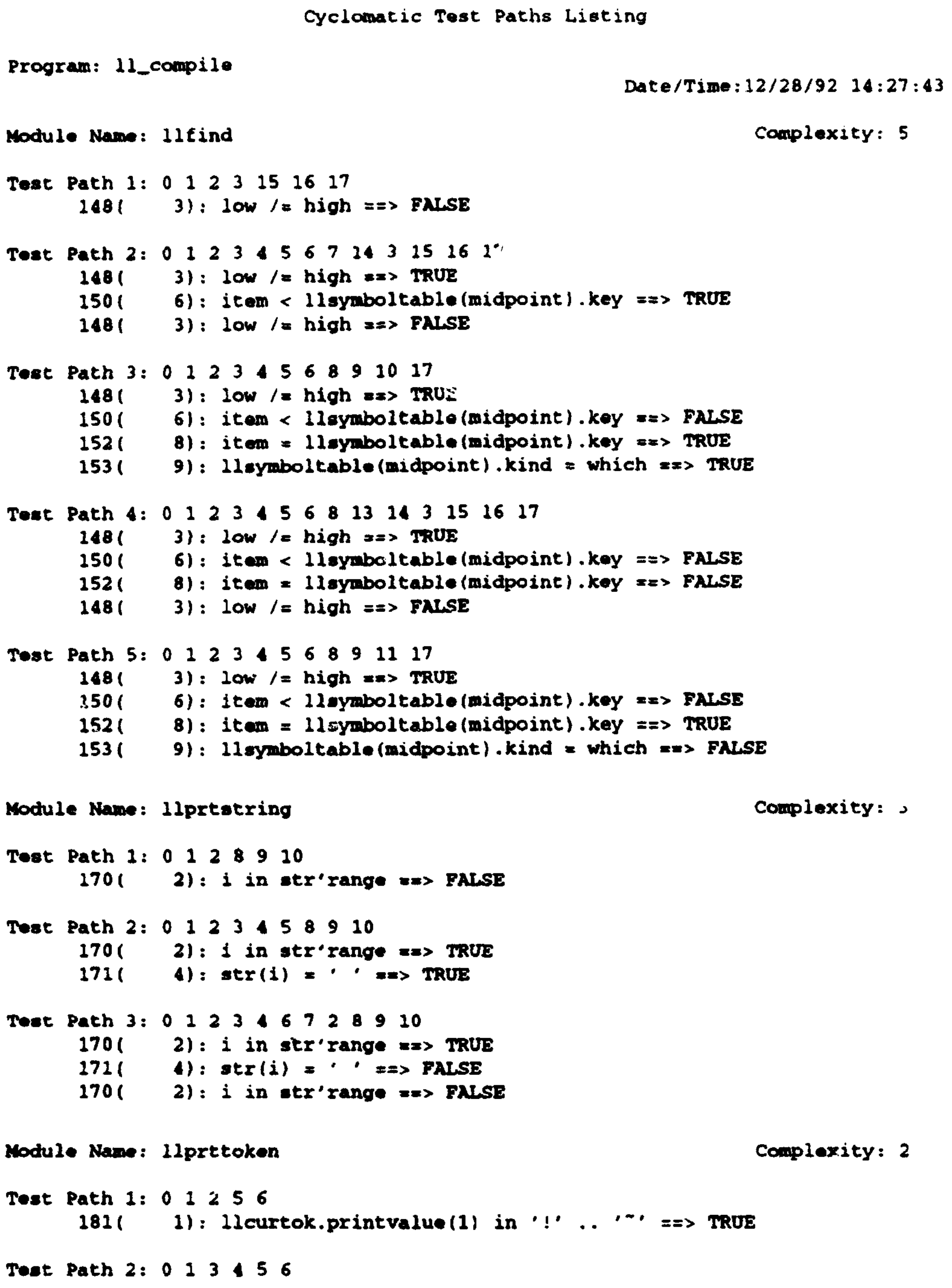
```
                              Program: ll_compile
                              -------------------

Module: llfind
    Program length (N): 74
    Program volume (V): 459
    Program level (L): 0.05
    Program difficulty (D): 21.5
    Intelligent content (I): 21.4
    Programming effort (E): 9879
    Error estimate (B): 0.15
    Programming time (T): 548.8

Module: llprtstring
    Program length (N): 37
    Program volume (V): 193
    Program level (L): 0.10
    Program difficulty (D): 10.5
    Intelligent content (I): 18.4
    Programming effort (E): 2024
    Error estimate (B): 0.06
    Programming time (T): 112.4

Module: llprttoken
    Program length (N): 42
    Program volume (V): 226
    Program level (L): 0.09
    Program difficulty (D): 11.5
    Intelligent content (I): 19.7
    Programming effort (E): 2604
    Error estimate (B): 0.08
    Programming time (T): 144.7

Module: llskiptoken
    Program length (N): 42
    Program volume (V): 226
    Program level (L): 0.08
    Program difficulty (D): 12.0
    Intelligent content (I): 18.9
    Programming effort (E): 2718
    Error estimate (B): 0.08
    Programming time (T): 151.0
                                    ...
Module: lltakeaction
    Program length (N): 1234
    Program volume (V): 12672
    Program level (L): 0.00
    Program difficulty (D): 323.5
    Intelligent content (I): 39.2
    Programming effort (E): 4099425
    Error estimate (B): 4.22
    Programming time (T): 227745.8
```

Figure 32-6. ACT Verbose Halstead's Metrics Report



```
                        Program: ll_compile
                        -------------------

        Module Name              Uniq Op    Uniq Opnd   Total Op   Total Opnd
-----------------------------------------------------------------------------
llfind                             43          31          43          31
llprtstring                        21          16          21          16
llprttoken                         23          19          23          19
llskiptoken                        24          18          24          18
llskipnode                         35          28          35          28
llskipboth                         37          29          37          29
llfatal                            22          16          22          16
get_character                      22          18          22          18
cvt_string                         18          14          18          14
make_token                        123          88         123          88
llnexttoken                        21          18          21          18
buildright                        135         116         135         116
buildselect                        23          19          23          19
readgram                           97          79          97          79
erase                              21          17          21          17
match                              28          19          28          19
expand;1                          102          90         102          90
synchronize                       143         107         143         107
testsynch                          20          15          20          15
parse                             133         106         133         106
llmain                              4           2           4           2
ll_compile                          2           1           2           1
get_char                           21          17          21          17
char_advance                       26          22          26          22
look_ahead                          9           8           9           8
next_character                     66          48          66          48
next_identifier                    50          36          50          36
next_spec_sym                      86          63          86          63
next_string                        45          34          45          34
advance                            66          48          66          48
merge_ranges                       22          18          22          18
alternate                         179         135         179         135
char_range                         31          24          31          24
restrict                          164         125         164         125
tail                              177         143         177         143
resolve_ambiguity                 380         305         380         305
complete_alt                      123         102         123         102
complete_concat                   123         102         123         102
complete_opt                       32          28          32          28
complete_pat                      128         100         128         100
complete_patterns                  11          10          11          10
concatenate                        27          18          27          18
cvt_ascii                          89          82          89          82
                                  ...
store_pattern                      65          57          65          57
lltakeaction                      647         587         647         587
```

Figure 32-7. ACT Halstead's Operand Count Report

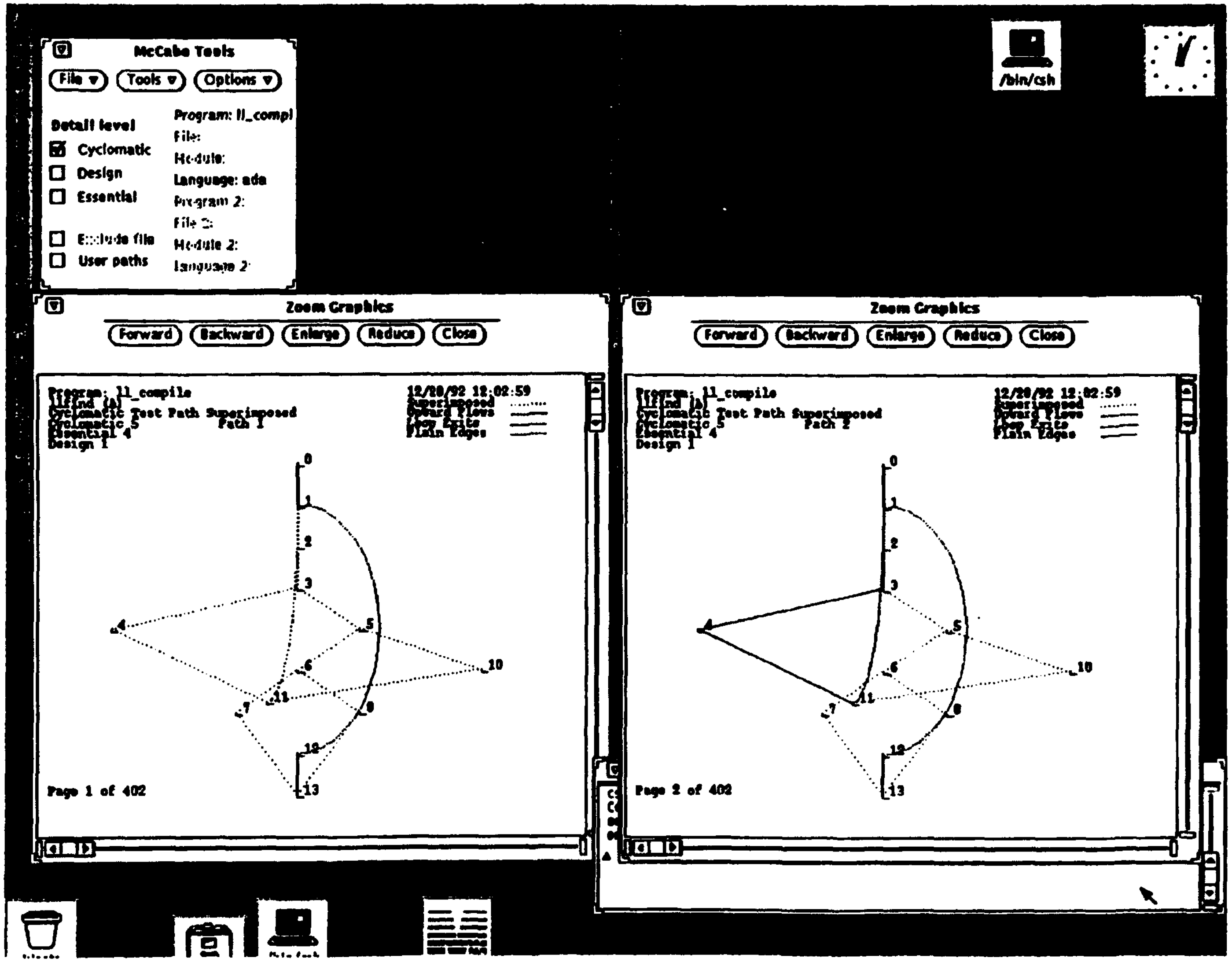


```
                    Cyclomatic Test Paths Listing

Program: ll_compile
                                             Date/Time:12/28/92 14:27:43

Module Name: llfind                                       Complexity: 5

Test Path 1: 0 1 2 3 15 16 17
      148(    3): low /= high ==> FALSE

Test Path 2: 0 1 2 3 4 5 6 7 14 3 15 16 17
      148(    3): low /= high ==> TRUE
      150(    6): item < llsymboltable(midpoint).key ==> TRUE
      148(    3): low /= high ==> FALSE

Test Path 3: 0 1 2 3 4 5 6 8 9 10 17
      148(    3): low /= high ==> TRUE
      150(    6): item < llsymboltable(midpoint).key ==> FALSE
      152(    8): item = llsymboltable(midpoint).key ==> TRUE
      153(    9): llsymboltable(midpoint).kind = which ==> TRUE

Test Path 4: 0 1 2 3 4 5 6 8 13 14 3 15 16 17
      148(    3): low /= high ==> TRUE
      150(    6): item < llsymboltable(midpoint).key ==> FALSE
      152(    8): item = llsymboltable(midpoint).key ==> FALSE
      148(    3): low /= high ==> FALSE

Test Path 5: 0 1 2 3 4 5 6 8 9 11 17
      148(    3): low /= high ==> TRUE
      150(    6): item < llsymboltable(midpoint).key ==> FALSE
      152(    8): item = llsymboltable(midpoint).key ==> TRUE
      153(    9): llsymboltable(midpoint).kind = which ==> FALSE

Module Name: llprtstring                                  Complexity: 3

Test Path 1: 0 1 2 8 9 10
      170(    2): i in str'range ==> FALSE

Test Path 2: 0 1 2 3 4 5 8 9 10
      170(    2): i in str'range ==> TRUE
      171(    4): str(i) = ' ' ==> TRUE

Test Path 3: 0 1 2 3 4 6 7 2 8 9 10
      170(    2): i in str'range ==> TRUE
      171(    4): str(i) = ' ' ==> FALSE
      170(    2): i in str'range ==> FALSE

Module Name: llprttoken                                   Complexity: 2

Test Path 1: 0 1 2 5 6
      181(    1): llcurtok.printvalue(1) in '!' .. '~' ==> TRUE

Test Path 2: 0 1 3 4 5 6
```

Figure 32-8. ACT Cyclomatic Test Paths Listing

```
      181(    1): llcurtok.printvalue(1) in '!' .. '~' ==> FALSE

Module Name: llskiptoken                                  Complexity: 1

Test Path 1: 0 1 2 3 4 5 6 7 8
                                 ...

Test Path 56: 0 1 136 137 165 166
       28(    1): caseindex ==> 55

Test Path 57: 0 1 138 139 165 166
       28(    1): caseindex ==> 56

Test Path 58: 0 1 140 141 142 165 166
       28(    1): caseindex ==> 57

Test Path 59: 0 1 143 144 165 166
       28(    1): caseindex ==> 58

Test Path 60: 0 1 145 146 147 165 166
       28(    1): caseindex ==> 59

Test Path 61: 0 1 148 149 165 166
       28(    1): caseindex ==> 60

Test Path 62: 0 1 150 151 152 165 166
       28(    1): caseindex ==> 61

Test Path 63: 0 1 153 154 165 166
       28(    1): caseindex ==> 62

Test Path 64: 0 1 155 156 165 166
       28(    1): caseindex ==> 63

Test Path 65: 0 1 157 158 165 166
       28(    1): caseindex ==> 64

Test Path 66: 0 1 159 160 165 166
       28(    1): caseindex ==> 65

Test Path 67: 0 1 161 162 165 166
       28(    1): caseindex ==> 66

Test Path 68: 0 1 163 164 165 166
       28(    1): caseindex ==> others
```

Figure 32-8. continued. ACT Cyclomatic Test Paths Listing



McCabe Tools

Detail level
☒ Cyclomatic
□ Design
□ Essential

□ Exclude file
□ User paths

Program: ll_compl
File:
Module:
Language: ada
Program 2:
File 2:
Module 2:
Language 2:

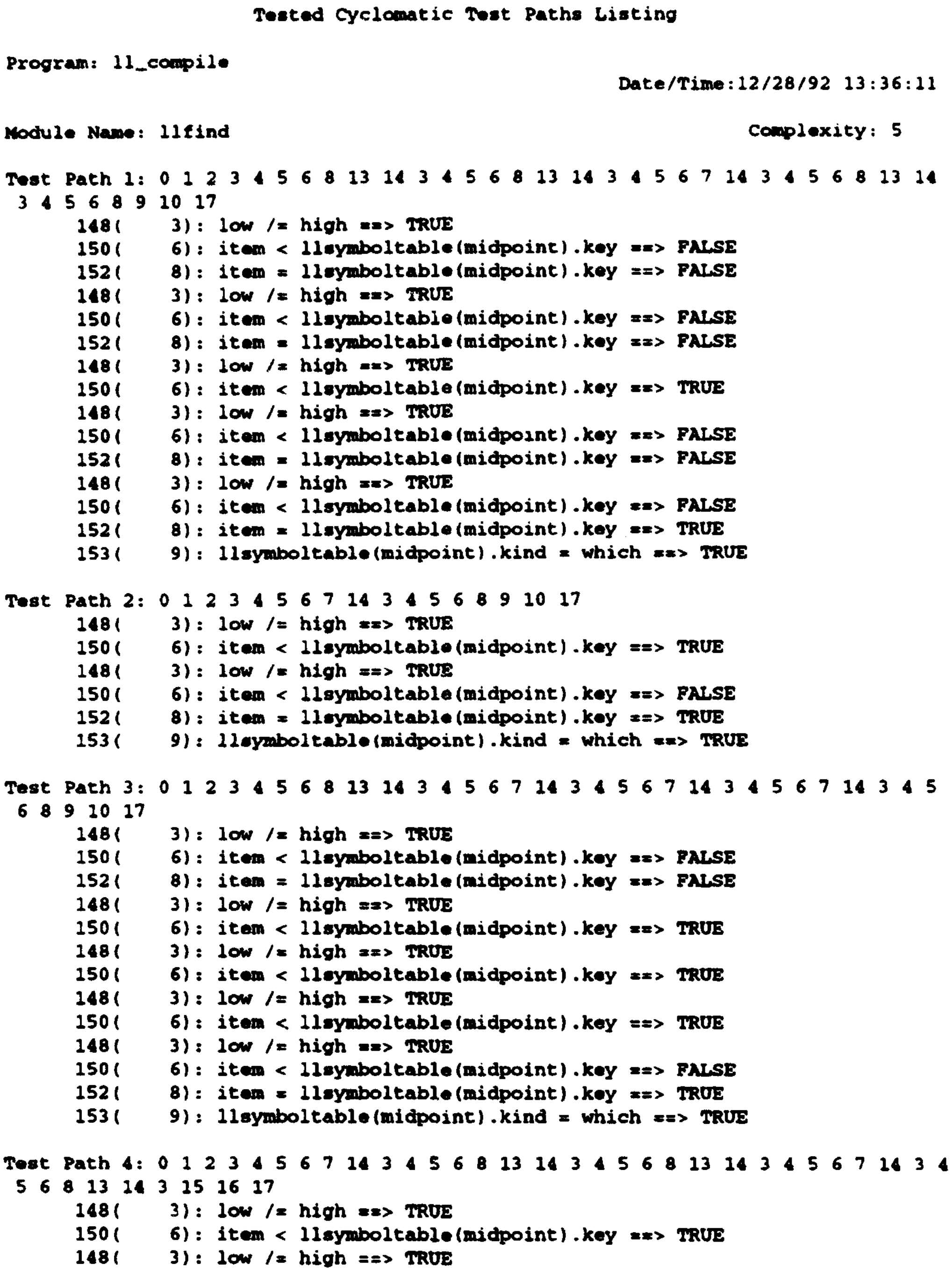
Zoom Graphics

```
Program: ll_compile                              12/20/92 12:02:59
llfind (a)                                       Superimposed  ........
Cyclomatic Test Path Superimposed                Upward Flows  ________
Cyclomatic 5                  Path 1             Loop Exits    ________
Essential 4                                      Plain Edges   ________
Design 1

Page 1 of 402
```

Zoom Graphics

```
Program: ll_compile                              12/20/92 12:02:59
llfind (a)                                       Superimposed  ........
Cyclomatic Test Path Superimposed                Upward Flows  ________
Cyclomatic 5                  Path 2             Loop Exits    ________
Essential 4                                      Plain Edges   ________
Design 1

Page 2 of 402
```

Figure 32-9. ACT Cyclomatic Test Paths Graph

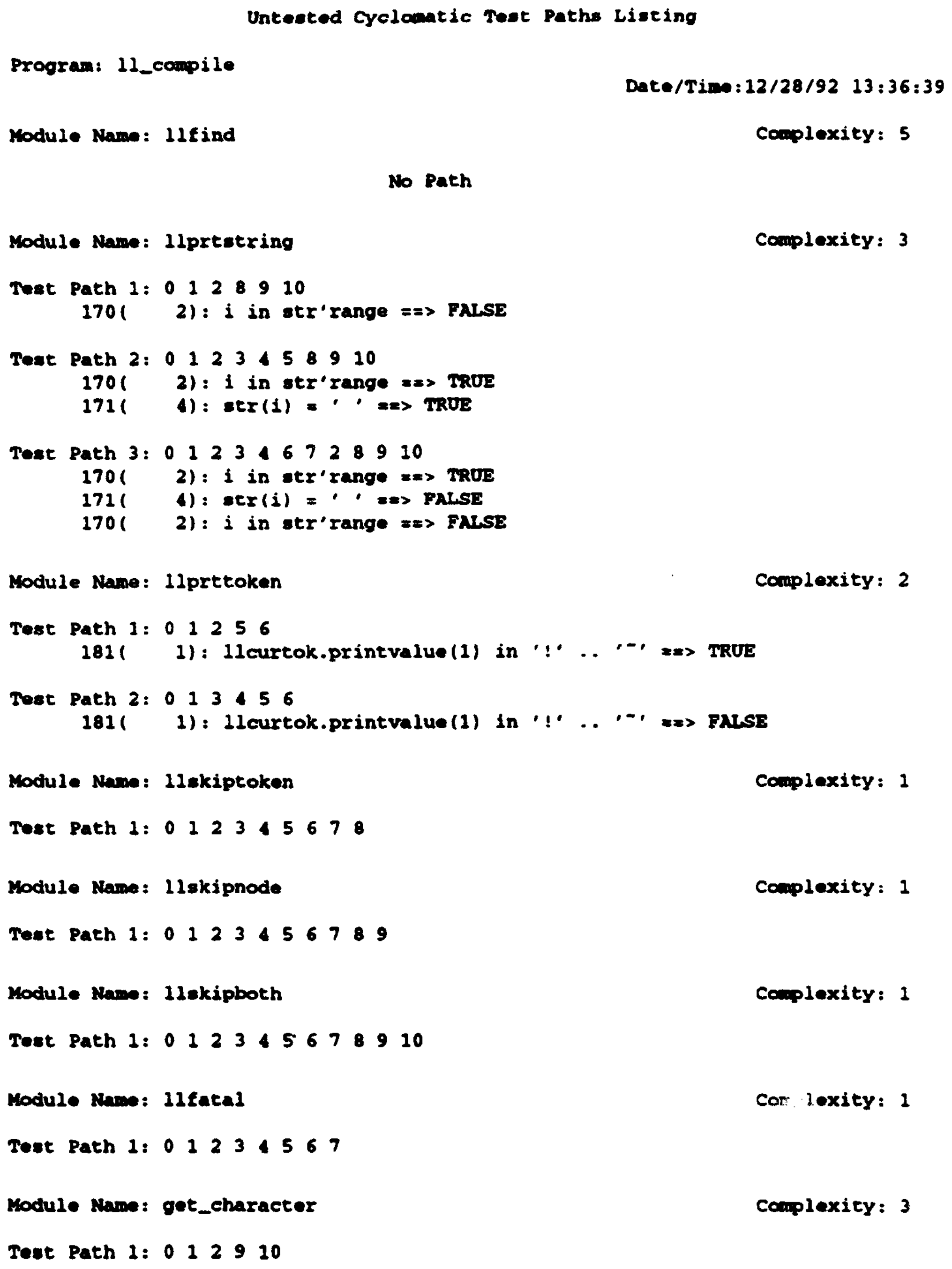


```
                    Tested Cyclomatic Test Paths Listing

Program: ll_compile
                                                      Date/Time:12/28/92 13:36:11

Module Name: llfind                                           Complexity: 5

Test Path 1: 0 1 2 3 4 5 6 8 13 14 3 4 5 6 8 13 14 3 4 5 6 7 14 3 4 5 6 8 13 14
 3 4 5 6 8 9 10 17
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> FALSE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> FALSE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> FALSE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> TRUE
      153(     9): llsymboltable(midpoint).kind = which ==> TRUE

Test Path 2: 0 1 2 3 4 5 6 7 14 3 4 5 6 8 9 10 17
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> TRUE
      153(     9): llsymboltable(midpoint).kind = which ==> TRUE

Test Path 3: 0 1 2 3 4 5 6 8 13 14 3 4 5 6 7 14 3 4 5 6 7 14 3 4 5 6 7 14 3 4 5
 6 8 9 10 17
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> FALSE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> FALSE
      152(     8): item = llsymboltable(midpoint).key ==> TRUE
      153(     9): llsymboltable(midpoint).kind = which ==> TRUE

Test Path 4: 0 1 2 3 4 5 6 7 14 3 4 5 6 8 13 14 3 4 5 6 8 13 14 3 4 5 6 7 14 3 4
 5 6 8 13 14 3 15 16 17
      148(     3): low /= high ==> TRUE
      150(     6): item < llsymboltable(midpoint).key ==> TRUE
      148(     3): low /= high ==> TRUE
```

Figure 32-10. ACT Tested Cyclomatic Paths Using test1.lex and sample.lex

```
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> FALSE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> FALSE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> TRUE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> FALSE
        148(    3): low /= high ==> FALSE

Test Path 5: 0 1 2 3 4 5 6 7 14 3 4 5 6 8 13 14 3 4 5 6 7 14 3 4 5 6 8 13 14 3 4
 5 6 8 9 11 17
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> TRUE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> FALSE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> TRUE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> FALSE
        148(    3): low /= high ==> TRUE
        150(    6): item < llsymboltable(midpoint).key ==> FALSE
        152(    8): item = llsymboltable(midpoint).key ==> TRUE
        153(    9): llsymboltable(midpoint).kind = which ==> FALSE


Module Name: llprtstring                                     Complexity: 3
                                No Path

Module Name: llprttoken                                      Complexity: 2
                                No Path
                                  ...

Test Path 31: 0 1 159 160 165 166
        28(     1): caseindex ==> 65

Test Path 32: 0 1 13 14 15 16 17 18 165 166
        28(     1): caseindex ==> 4

Test Path 33: 0 1 83 84 85 165 166
        28(     1): caseindex ==> 33

Test Path 34: 0 1 62 63 64 165 166
        28(     1): caseindex ==> 25

Test Path 35: 0 1 128 129 130 131 165 166
        28(     1): caseindex ==> 52
```

Figure 32-10. continued. ACT Tested Cyclomatic Paths using test1.lex and sample.lex

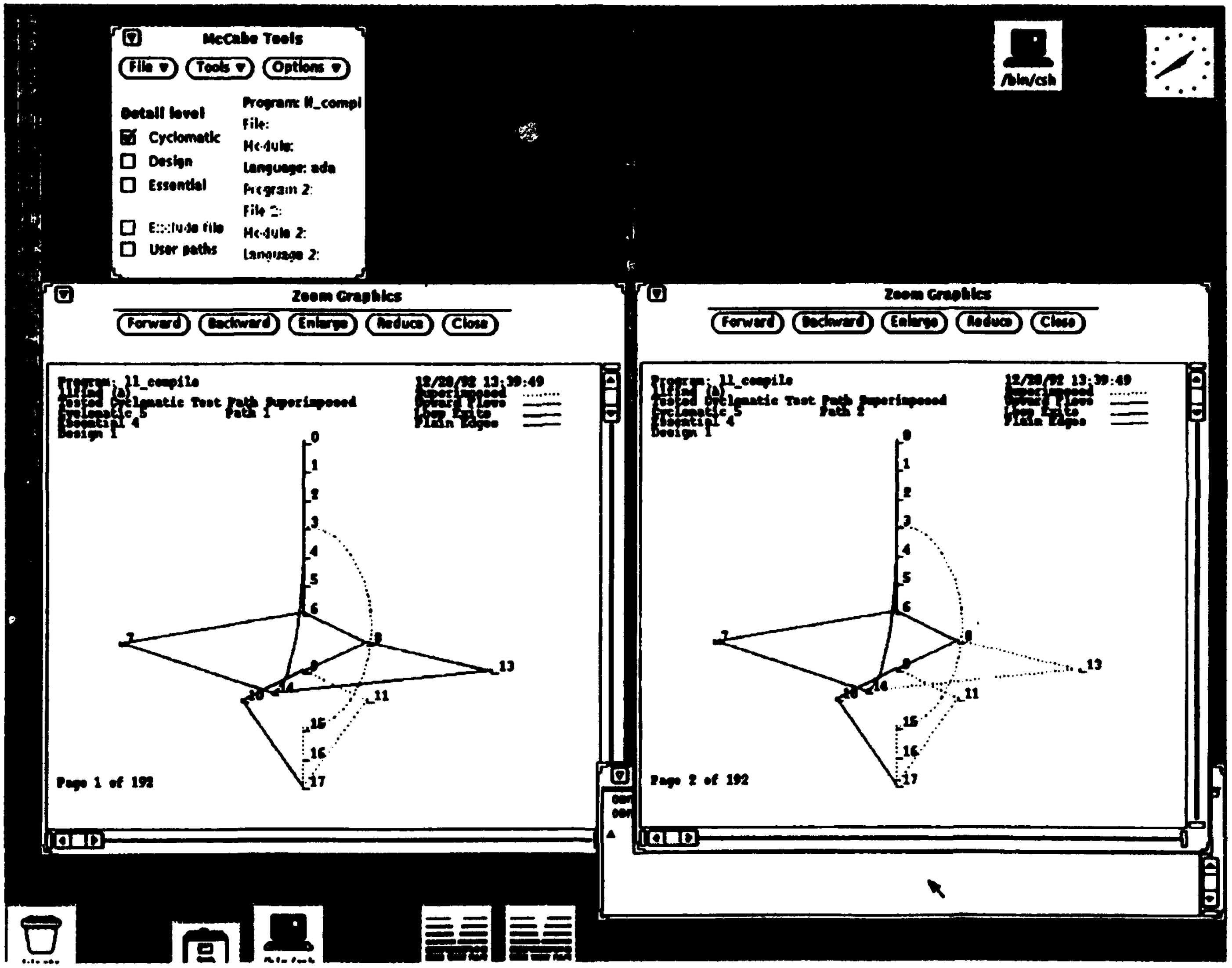


```
                        Untested Cyclomatic Test Paths Listing

Program: ll_compile
                                                       Date/Time:12/28/92 13:36:39

Module Name: llfind                                              Complexity: 5

                                   No Path

Module Name: llprtstring                                         Complexity: 3

Test Path 1: 0 1 2 8 9 10
      170(    2): i in str'range ==> FALSE

Test Path 2: 0 1 2 3 4 5 8 9 10
      170(    2): i in str'range ==> TRUE
      171(    4): str(i) = ' ' ==> TRUE

Test Path 3: 0 1 2 3 4 6 7 2 8 9 10
      170(    2): i in str'range ==> TRUE
      171(    4): str(i) = ' ' ==> FALSE
      170(    2): i in str'range ==> FALSE

Module Name: llprttoken                                          Complexity: 2

Test Path 1: 0 1 2 5 6
      181(    1): llcurtok.printvalue(1) in '!' .. '~' ==> TRUE

Test Path 2: 0 1 3 4 5 6
      181(    1): llcurtok.printvalue(1) in '!' .. '~' ==> FALSE

Module Name: llskiptoken                                         Complexity: 1

Test Path 1: 0 1 2 3 4 5 6 7 8

Module Name: llskipnode                                          Complexity: 1

Test Path 1: 0 1 2 3 4 5 6 7 8 9

Module Name: llskipboth                                          Complexity: 1

Test Path 1: 0 1 2 3 4 5 6 7 8 9 10

Module Name: llfatal                                             Complexity: 1

Test Path 1: 0 1 2 3 4 5 6 7

Module Name: get_character                                       Complexity: 3

Test Path 1: 0 1 2 9 10
```

Figure 32-11. ACT Untested Cyclomatic Paths Using test1.lex and sample.lex

```
      252(    1): end_of_file(standard_input) ==> TRUE

Test Path 2: 0 1 3 7 8 9 10
      252(    1): end_of_file(standard_input) ==> FALSE
      254(    3): end_of_line(standard_input) ==> FALSE

Test Path 3: 0 1 3 4 5 6 9 10
      252(    1): end_of_file(standard_input) ==> FALSE
      254(    3): end_of_line(standard_input) ==> TRUE

Module Name: cvt_string                                    Complexity: 3

Test Path 1: 0 1 7 8 9
      276(    1): i in llstrings'range ==> FALSE

Test Path 2: 0 1 2 3 4 6 1 7 8 9
      276(    1): i in llstrings'range ==> TRUE
      277(    3): i <= str'last ==> TRUE
      276(    1): i in llstrings'range ==> FALSE

Test Path 3: 0 1 2 3 5 6 1 7 8 9
      276(    1): i in llstrings'range ==> TRUE
      277(    3): i <= str'last ==> FALSE
      276(    1): i in llstrings'range ==> FALSE

Module Name: make_token                                    Complexity: 11

Test Path 1: 0 1 2 3 24 25 26 27 28 29 38 39 40
      288(    3): node ==> others
      306(   27): node ==> char

Test Path 2: 0 1 2 3 4 5 6 26 27 28 29 38 39 40
      288(    3): node ==> char
      306(   27): node ==> char
                           ...

Test Path 28: 0 1 148 149 165 166
       28(    1): caseindex ==> 60

Test Path 29: 0 1 150 151 152 165 166
       28(    1): caseindex ==> 61

Test Path 30: 0 1 153 154 165 166
       28(    1): caseindex ==> 62

Test Path 31: 0 1 155 156 165 166
       28(    1): caseindex ==> 63

Test Path 32: 0 1 161 162 165 166
       28(    1): caseindex ==> 66

Test Path 33: 0 1 163 164 165 166
       28(    1): caseindex ==> others
```

Figure 32-11. continued. ACT Untested Cyclomatic Paths Using test1.lex and sample.lex



Detail level
☒ Cyclomatic
☐ Design
☐ Essential

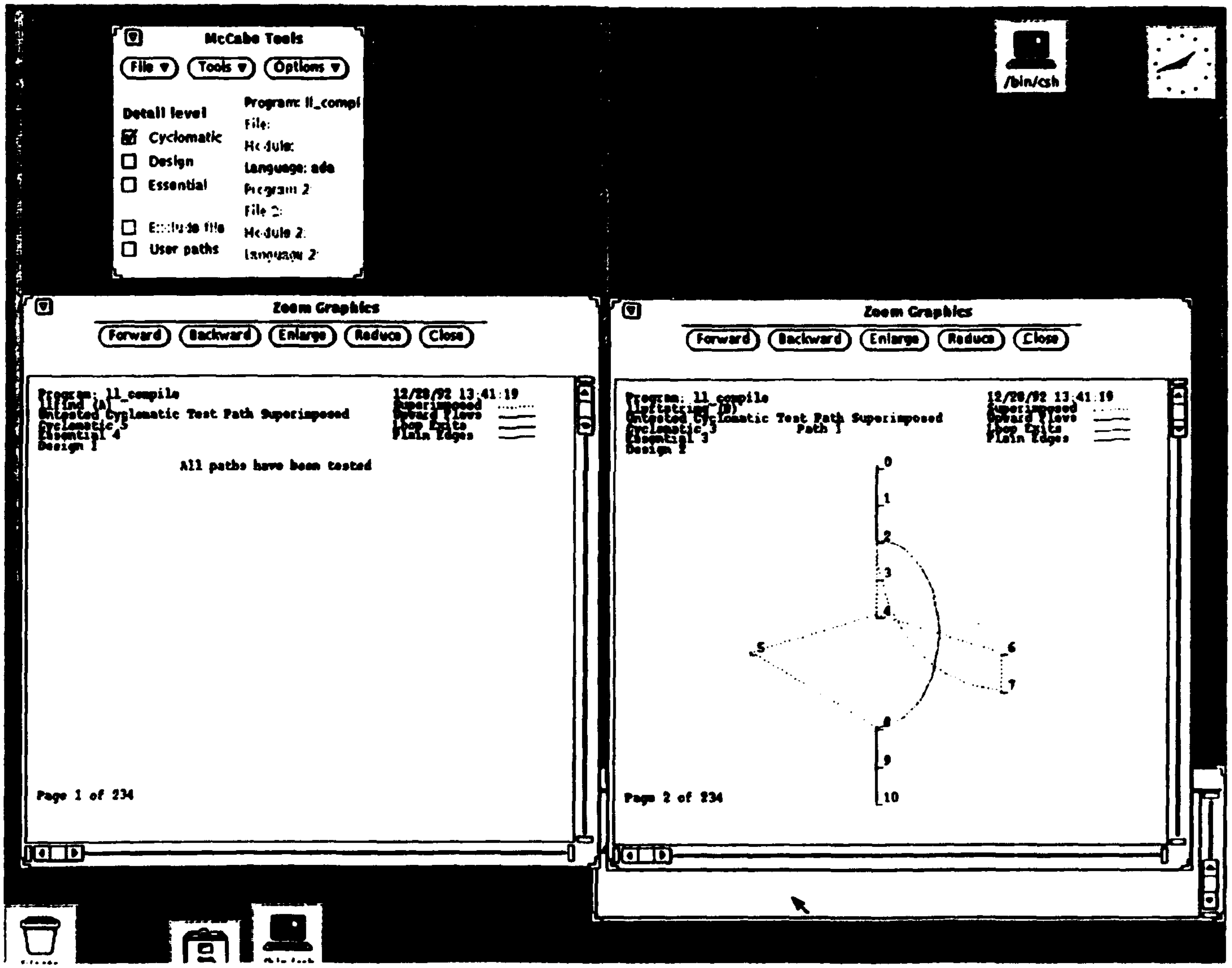
☐ Exclude file
☐ User paths

Program: 11_compl
File:
Module:
Language: ada
Program 2:
File 2:
Module 2:
Language 2:

```
Program: 11_compile                                   12/28/92 13:39:49
[illegible] (a)                                       Superimposed    ........
Tested Cyclomatic Test Path Superimposed              Upward Flows    ______
Cyclomatic 5            Path 1                        Loop Exits      ______
Essential 4                                           Plain Edges     ______
Design 1

Page 1 of 192
```

```
Program: 11_compile                                   12/28/92 13:39:49
[illegible] (a)                                       Superimposed    ........
Tested Cyclomatic Test Path Superimposed              Upward Flows    ______
Cyclomatic 5            Path 2                        Loop Exits      ______
Essential 4                                           Plain Edges     ______
Design 1

Page 2 of 192
```

Figure 32-12. ACT Tested Paths Graph Using test1.lex and sample.lex



Detail level
☒ Cyclomatic
☐ Design
☐ Essential

☐ Exclude file
☐ User paths

Program: ll_compl
File:
Module:
Language: ada
Program 2:
File 2:
Module 2:
Language 2:

```
Program: ll_compile                                  12/28/92 13:41:19
llfind (A)                                           Superimposed   ........
Untested Cyclomatic Test Path Superimposed           Upward Flows   ————
Cyclomatic 5                                         Loop Exits     ————
Essential 4                                          Plain Edges    ————
Design 1
                 All paths have been tested

Page 1 of 234
```

```
Program: ll_compile                                  12/28/92 13:41:19
llftstring (B)                                       Superimposed   ........
Untested Cyclomatic Test Path Superimposed           Upward Flows   ————
Cyclomatic 3              Path 1                     Loop Exits     ————
Essential 3                                          Plain Edges    ————
Design 2

Page 2 of 234
```

Figure 32-13. ACT Untested Paths Graph Using test1.lex and sample.lex

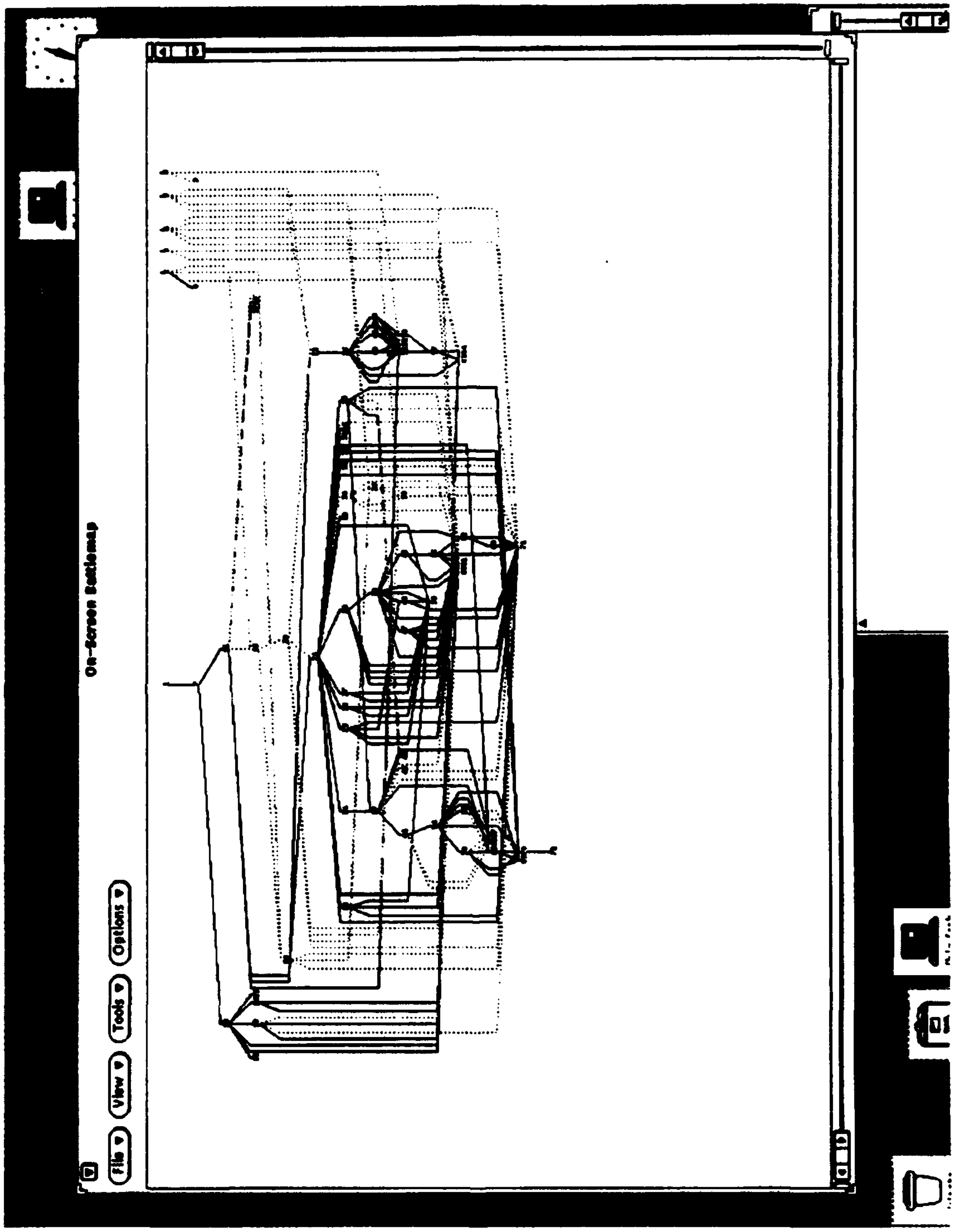
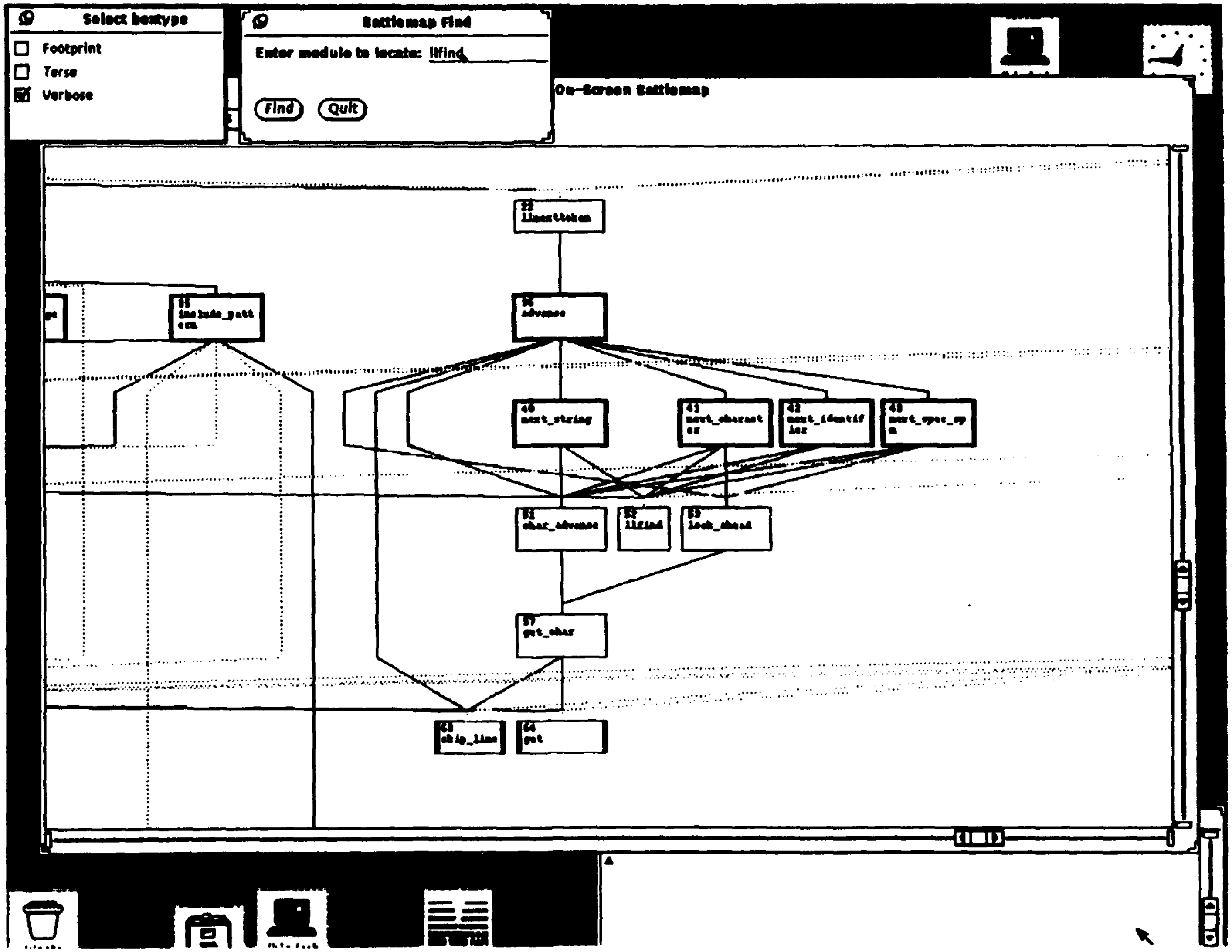


Figure 32-14. BAT Battlemap for Program LL_COMPILE



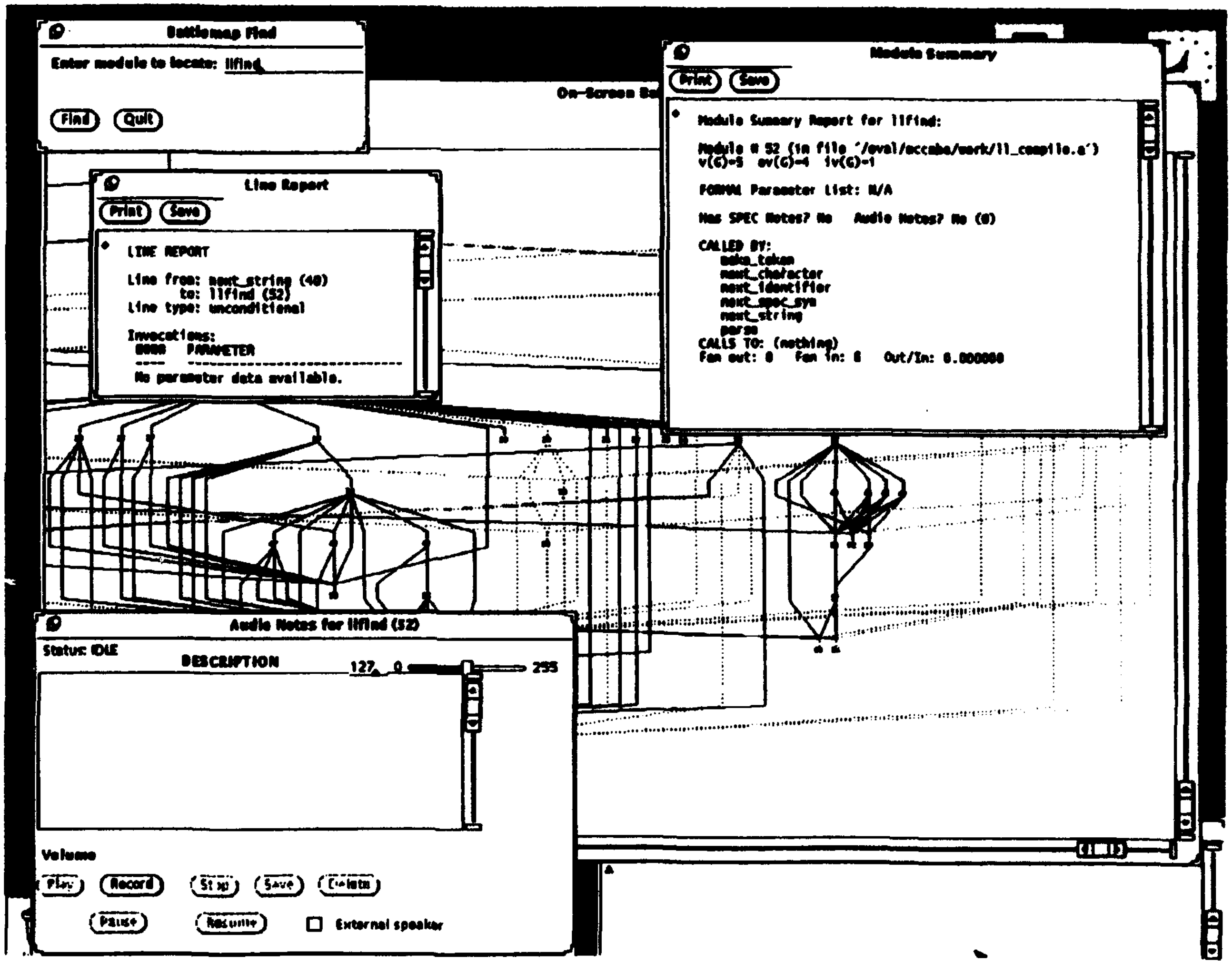
Select boxtype

□ Footprint
□ Terse
☑ Verbose

Battlemap Find

Enter module to locate: llfind

Figure 32-15. BAT Battlemap Symbology



Enter module to locate: llfind

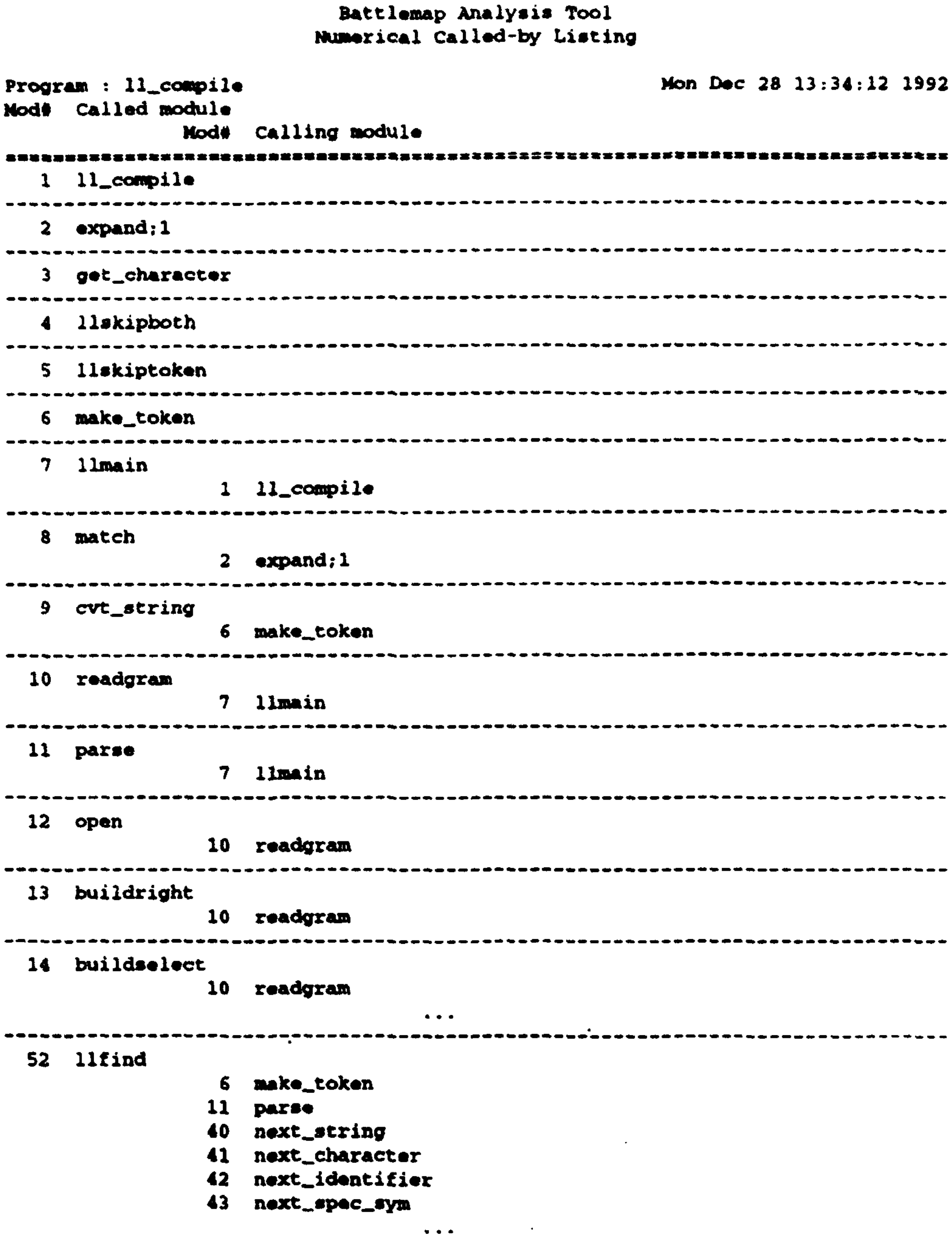
```
LINE REPORT

Line from: next_string (40)
       to: llfind (52)
Line type: unconditional

Invocations:
 NNNN   PARAMETER
 ----   ------------------------------
 No parameter data available.
```

```
Module Summary Report for llfind:

Module # 52 (in file '/eval/mccabe/work/ll_compile.a')
v(G)=5  ev(G)=4  iv(G)=1

FORMAL Parameter List: N/A

Has SPEC Notes? No   Audio Notes? No (0)

CALLED BY:
   make_token
   next_character
   next_identifier
   next_spec_sym
   next_string
   parse
CALLS TO: (nothing)
Fan out: 0   Fan in: 6   Out/In: 0.000000
```

Status: IDLE

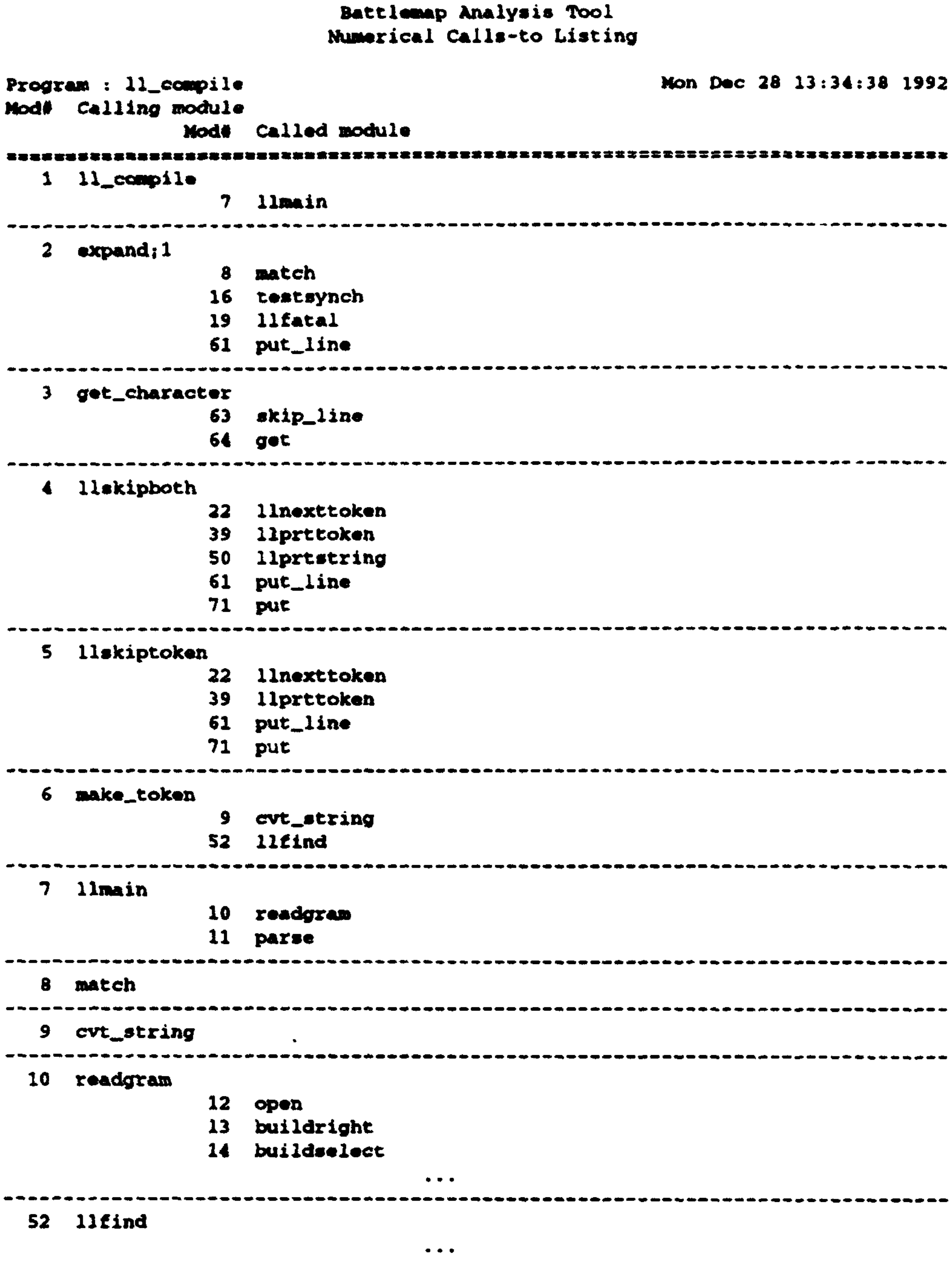
□ External speaker

Figure 32-16. BAT Searching and Summaries



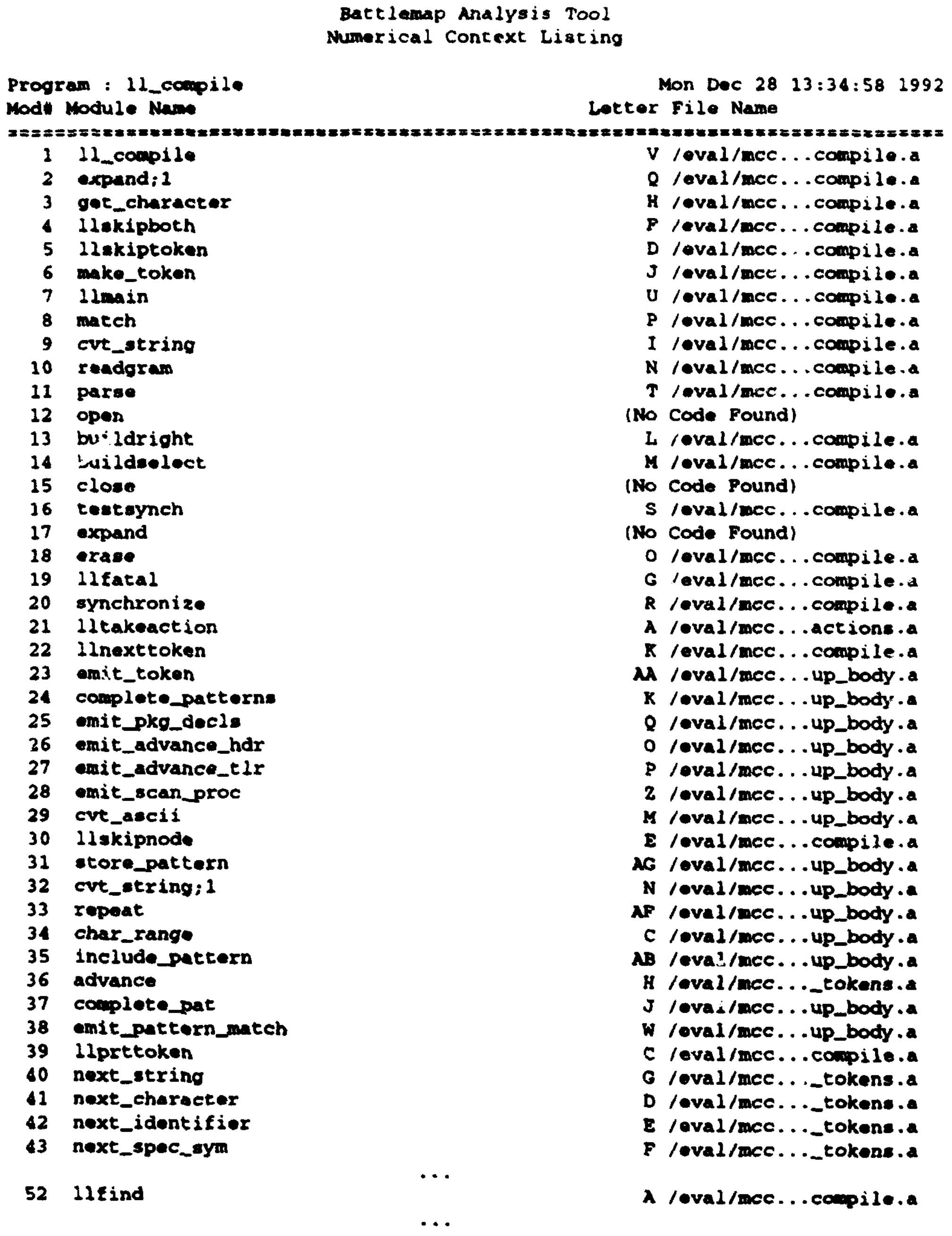
```
                        Battlemap Analysis Tool
                      Numerical Called-by Listing

Program : ll_compile                            Mon Dec 28 13:34:12 1992
Mod#  Called module
              Mod#  Calling module
==========================================================================
  1  ll_compile
--------------------------------------------------------------------------
  2  expand;1
--------------------------------------------------------------------------
  3  get_character
--------------------------------------------------------------------------
  4  llskipboth
--------------------------------------------------------------------------
  5  llskiptoken
--------------------------------------------------------------------------
  6  make_token
--------------------------------------------------------------------------
  7  llmain
                 1  ll_compile
--------------------------------------------------------------------------
  8  match
                 2  expand;1
--------------------------------------------------------------------------
  9  cvt_string
                 6  make_token
--------------------------------------------------------------------------
 10  readgram
                 7  llmain
--------------------------------------------------------------------------
 11  parse
                 7  llmain
--------------------------------------------------------------------------
 12  open
                10  readgram
--------------------------------------------------------------------------
 13  buildright
                10  readgram
--------------------------------------------------------------------------
 14  buildselect
                10  readgram
                              ...
--------------------------------------------------------------------------
 52  llfind
                 6  make_token
                11  parse
                40  next_string
                41  next_character
                42  next_identifier
                43  next_spec_sym
                              ...
```

Figure 32-17. BAT Numerical Called-by Listing



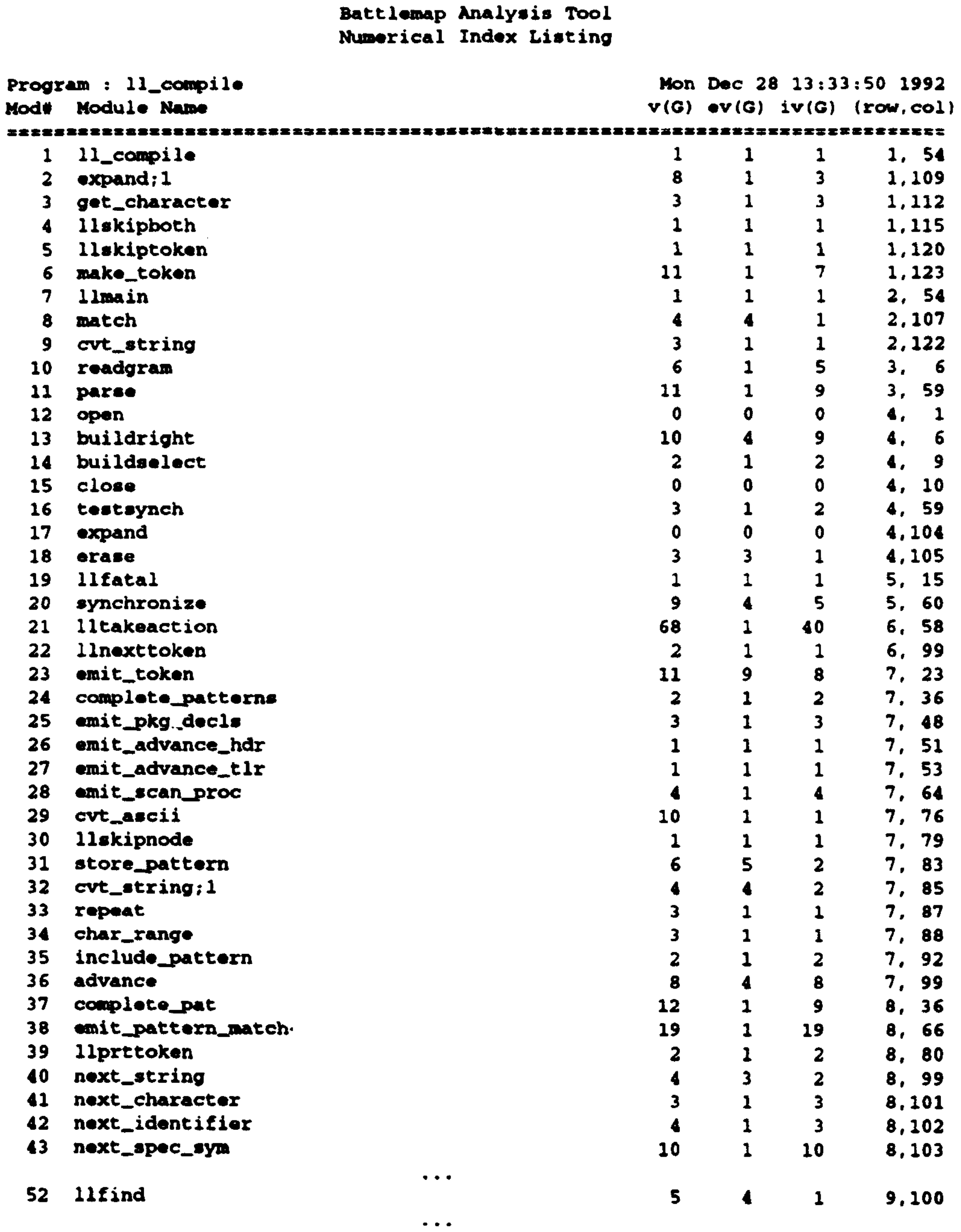
```
                          Battlemap Analysis Tool
                        Numerical Calls-to Listing

Program : ll_compile                                 Mon Dec 28 13:34:38 1992
Mod#  Calling module
                Mod#  Called module
==========================================================================
  1  ll_compile
                   7  llmain
--------------------------------------------------------------------------
  2  expand;1
                   8  match
                  16  testsynch
                  19  llfatal
                  61  put_line
--------------------------------------------------------------------------
  3  get_character
                  63  skip_line
                  64  get
--------------------------------------------------------------------------
  4  llskipboth
                  22  llnexttoken
                  39  llprttoken
                  50  llprtstring
                  61  put_line
                  71  put
--------------------------------------------------------------------------
  5  llskiptoken
                  22  llnexttoken
                  39  llprttoken
                  61  put_line
                  71  put
--------------------------------------------------------------------------
  6  make_token
                   9  cvt_string
                  52  llfind
--------------------------------------------------------------------------
  7  llmain
                  10  readgram
                  11  parse
--------------------------------------------------------------------------
  8  match
--------------------------------------------------------------------------
  9  cvt_string
--------------------------------------------------------------------------
 10  readgram
                  12  open
                  13  buildright
                  14  buildselect
                                ...
--------------------------------------------------------------------------
 52  llfind
                                ...
```

Figure 32-18. BAT Numerical Calls-to Listing



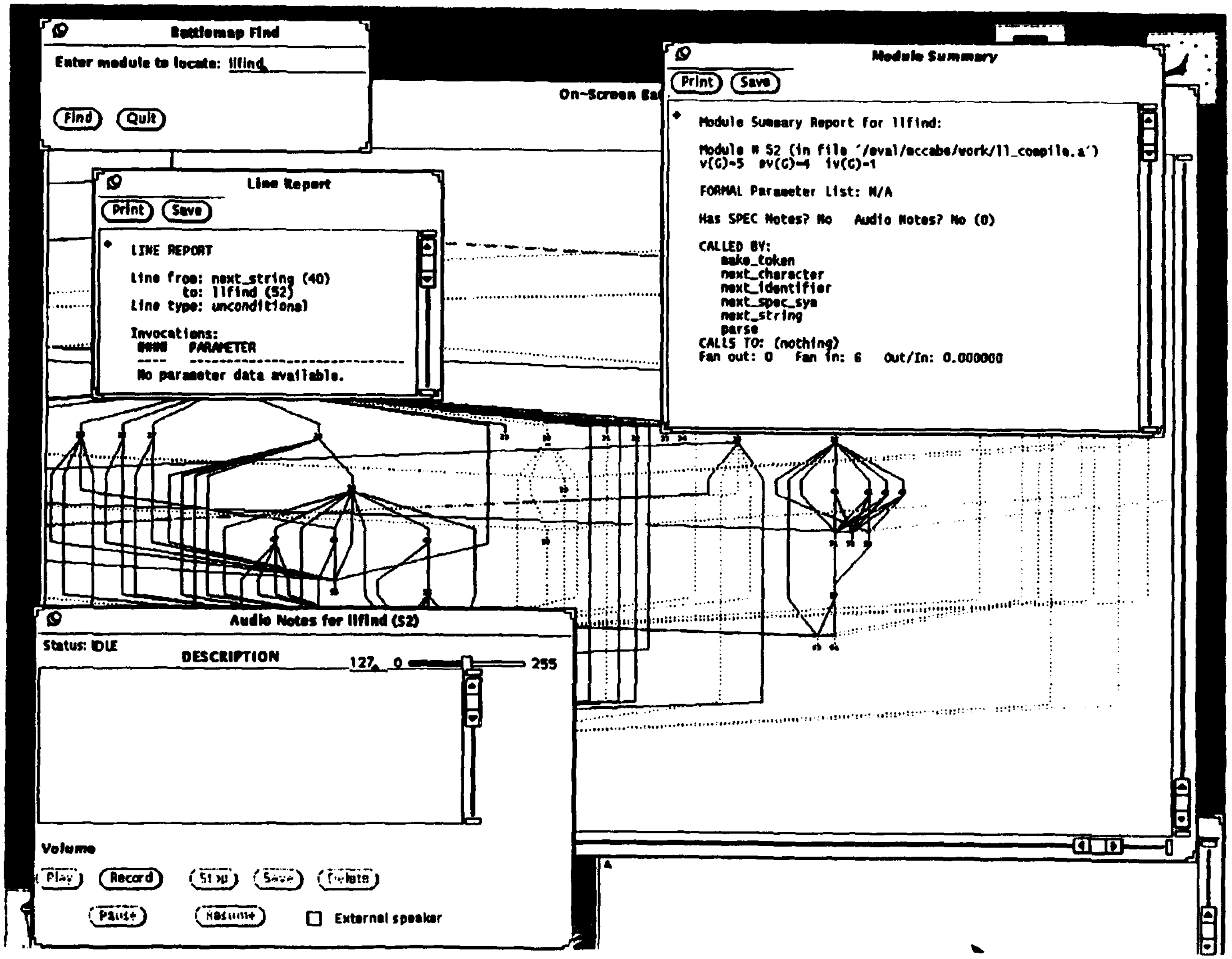
```
                     Battlemap Analysis Tool
                    Numerical Context Listing

Program : ll_compile                              Mon Dec 28 13:34:58 1992
Mod# Module Name                               Letter File Name
==========================================================================
   1  ll_compile                                    V /eval/mcc...compile.a
   2  expand;1                                      Q /eval/mcc...compile.a
   3  get_character                                 H /eval/mcc...compile.a
   4  llskipboth                                    F /eval/mcc...compile.a
   5  llskiptoken                                   D /eval/mcc...compile.a
   6  make_token                                    J /eval/mcc...compile.a
   7  llmain                                        U /eval/mcc...compile.a
   8  match                                         P /eval/mcc...compile.a
   9  cvt_string                                    I /eval/mcc...compile.a
  10  readgram                                      N /eval/mcc...compile.a
  11  parse                                         T /eval/mcc...compile.a
  12  open                                      (No Code Found)
  13  buildright                                    L /eval/mcc...compile.a
  14  buildselect                                   M /eval/mcc...compile.a
  15  close                                     (No Code Found)
  16  testsynch                                     S /eval/mcc...compile.a
  17  expand                                    (No Code Found)
  18  erase                                         O /eval/mcc...compile.a
  19  llfatal                                       G /eval/mcc...compile.a
  20  synchronize                                   R /eval/mcc...compile.a
  21  lltakeaction                                  A /eval/mcc...actions.a
  22  llnexttoken                                   K /eval/mcc...compile.a
  23  emit_token                                   AA /eval/mcc...up_body.a
  24  complete_patterns                             K /eval/mcc...up_body.a
  25  emit_pkg_decls                                Q /eval/mcc...up_body.a
  26  emit_advance_hdr                              O /eval/mcc...up_body.a
  27  emit_advance_tlr                              P /eval/mcc...up_body.a
  28  emit_scan_proc                                Z /eval/mcc...up_body.a
  29  cvt_ascii                                     M /eval/mcc...up_body.a
  30  llskipnode                                    E /eval/mcc...compile.a
  31  store_pattern                                AG /eval/mcc...up_body.a
  32  cvt_string;1                                  N /eval/mcc...up_body.a
  33  repeat                                       AF /eval/mcc...up_body.a
  34  char_range                                    C /eval/mcc...up_body.a
  35  include_pattern                              AB /eval/mcc...up_body.a
  36  advance                                       H /eval/mcc..._tokens.a
  37  complete_pat                                  J /eval/mcc...up_body.a
  38  emit_pattern_match                            W /eval/mcc...up_body.a
  39  llprttoken                                    C /eval/mcc...compile.a
  40  next_string                                   G /eval/mcc..._tokens.a
  41  next_character                                D /eval/mcc..._tokens.a
  42  next_identifier                               E /eval/mcc..._tokens.a
  43  next_spec_sym                                 F /eval/mcc..._tokens.a
                                 ...
  52  llfind                                        A /eval/mcc...compile.a
                                 ...
```

Figure 32-19. BAT Numerical Context Listing



```
                       Battlemap Analysis Tool
                       Numerical Index Listing

Program : ll_compile                          Mon Dec 28 13:33:50 1992
Mod#  Module Name                            v(G) ev(G) iv(G) (row,col)
========================================================================
  1   ll_compile                               1     1     1     1, 54
  2   expand;1                                 8     1     3     1,109
  3   get_character                            3     1     3     1,112
  4   llskipboth                               1     1     1     1,115
  5   llskiptoken                              1     1     1     1,120
  6   make_token                              11     1     7     1,123
  7   llmain                                   1     1     1     2, 54
  8   match                                    4     4     1     2,107
  9   cvt_string                               3     1     1     2,122
 10   readgram                                 6     1     5     3,  6
 11   parse                                   11     1     9     3, 59
 12   open                                     0     0     0     4,  1
 13   buildright                              10     4     9     4,  6
 14   buildselect                              2     1     2     4,  9
 15   close                                    0     0     0     4, 10
 16   testsynch                                3     1     2     4, 59
 17   expand                                   0     0     0     4,104
 18   erase                                    3     3     1     4,105
 19   llfatal                                  1     1     1     5, 15
 20   synchronize                              9     4     5     5, 60
 21   lltakeaction                            68     1    40     6, 58
 22   llnexttoken                              2     1     1     6, 99
 23   emit_token                              11     9     8     7, 23
 24   complete_patterns                        2     1     2     7, 36
 25   emit_pkg_decls                           3     1     3     7, 48
 26   emit_advance_hdr                         1     1     1     7, 51
 27   emit_advance_tlr                         1     1     1     7, 53
 28   emit_scan_proc                           4     1     4     7, 64
 29   cvt_ascii                               10     1     1     7, 76
 30   llskipnode                               1     1     1     7, 79
 31   store_pattern                            6     5     2     7, 83
 32   cvt_string;1                             4     4     2     7, 85
 33   repeat                                   3     1     1     7, 87
 34   char_range                               3     1     1     7, 88
 35   include_pattern                          2     1     2     7, 92
 36   advance                                  8     4     8     7, 99
 37   complete_pat                            12     1     9     8, 36
 38   emit_pattern_match                      19     1    19     8, 66
 39   llprttoken                               2     1     2     8, 80
 40   next_string                              4     3     2     8, 99
 41   next_character                           3     1     3     8,101
 42   next_identifier                          4     1     3     8,102
 43   next_spec_sym                           10     1    10     8,103
                              ...
 52   llfind                                   5     4     1     9,100
                              ...
```

Figure 32-20. BAT Numerical Index Listing



Enter module to locate: llfind

```
LINE REPORT

Line from: next_string (40)
       to: llfind (52)
Line type: unconditional

Invocations:
 ####   PARAMETER
 ----   ----------------------------
 No parameter data available.
```

```
Module Summary Report for llfind:

Module # 52 (in file '/eval/mccabe/work/ll_compile.a')
v(G)=5  ev(G)=4  iv(G)=1

FORMAL Parameter List: N/A

Has SPEC Notes? No   Audio Notes? No (0)

CALLED BY:
   make_token
   next_character
   next_identifier
   next_spec_sym
   next_string
   parse
CALLS TO: (nothing)
Fan out: 0   Fan in: 6   Out/In: 0.000000
```

Status: IDLE

DESCRIPTION 127 0 255

□ External speaker

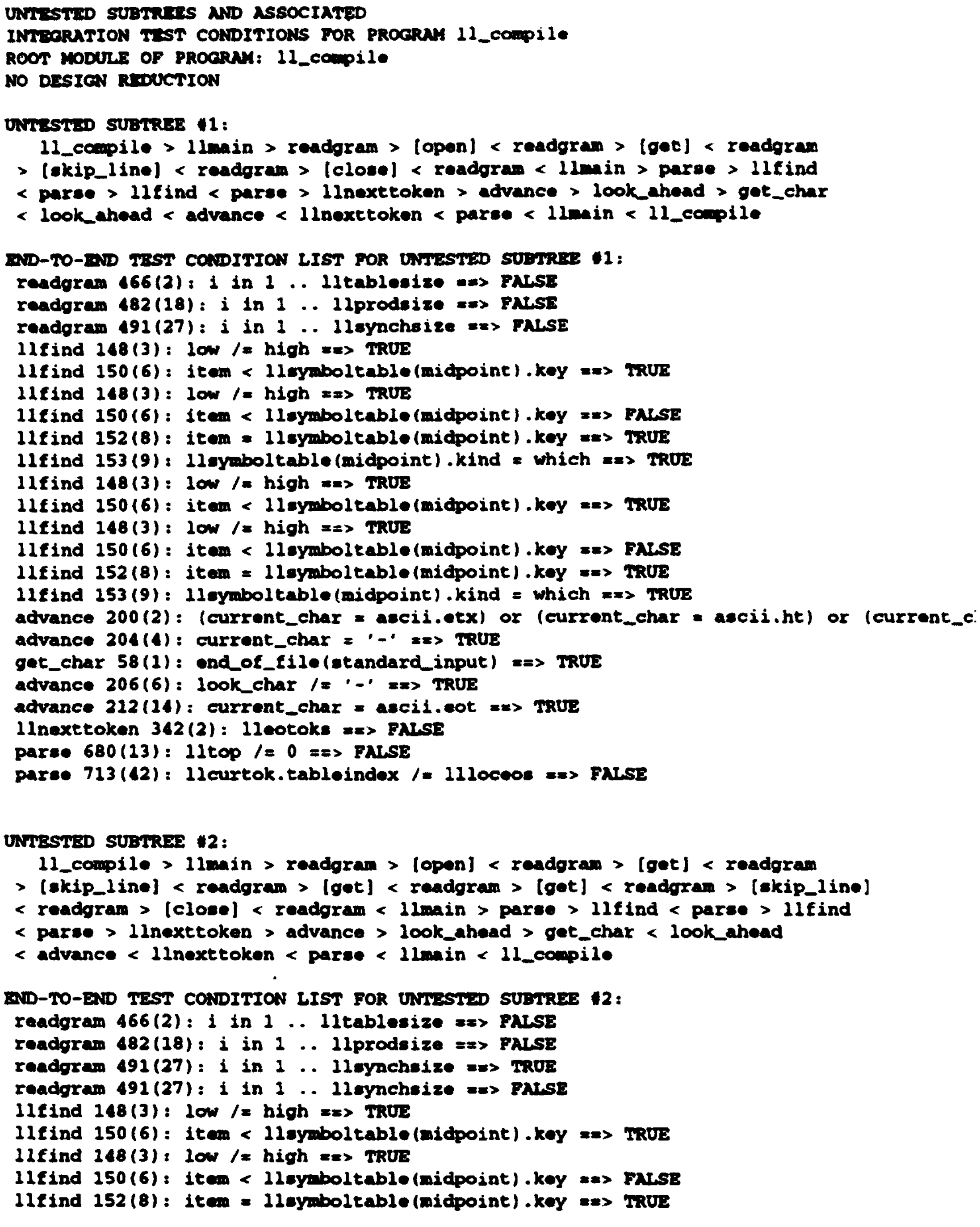
Figure 32-21. BAT Subtrees Graph

```
SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM ll_compile
ROOT MODULE OF PROGRAM: ll_compile
NO DESIGN REDUCTION

SUBTREE #1:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance < llnexttoken < parse
 < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #1:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #2:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [skip_line] < readgram
 > [close] < readgram < llmain > parse > llfind < parse > llfind < parse
 > llnexttoken > advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #2:
 readgram 466(2): i in 1 .. lltablesize ==> TRUE
 readgram 467(4): j in 1 .. llstringlength ==> FALSE
 readgram 472(10): ch = 'g' ==> TRUE
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #3:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [get] < readgram > [get]
 < readgram > [skip_line] < readgram > buildright < readgram > buildselect
 > [skip_line] < buildselect < readgram > [close] < readgram < llmain > parse
 > llfind < parse > llfind < parse > llnexttoken > advance < llnexttoken
 < parse < llmain < ll_compile
```

Figure 32-22. BAT Cyclomatic Subtrees

```
END-TO-END TEST CONDITION LIST FOR SUBTREE #3:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> TRUE
 buildright 397(3): i in thisrhs + 1 .. thisrhs + productions(whichprod).cardrhs ==>
 buildselect 455(2): i in 1 .. productions(whichprod).cardsel ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #4:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 ...


SUBTREE #315:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance < llnexttoken < parse
 > testsynch > synchronize > [put] < synchronize > llprttoken > llprtstring
 > [put] < llprtstring > [put] < llprtstring < llprttoken < synchronize
 > [put] < synchronize > [put] < synchronize > [put_line] < synchronize
 < testsynch < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #315:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> TRUE
 parse 682(16): llstack(llsentptr).data.kind ==> group | literal
 parse 684(17): llstack(llsentptr).data.tableindex = llcurtok.tableindex ==> FALSE
 parse 688(20): llstack(llsentptr).data.tableindex = locofnull ==> FALSE
 parse 690(22): not llstack(llsentptr).lastchild ==> TRUE
 parse 691(23): llstack(llsentptr + 1).data.kind = patch ==> FALSE
 testsynch 654(1): llstack(llsentptr).data.synchindex = 0 ==> FALSE
 llprttoken 181(1): llcurtok.printvalue(1) in '!' .. '~' ==> TRUE
 llprtstring 170(2): i in str'range ==> FALSE
 synchronize 610(8): synchdata(i).sent /= 0 ==> FALSE
 synchronize 643(36): llcurtok.tableindex = llloceos ==> TRUE
 parse 706(37): lladvance ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE
```

Figure 32-22. continued. BAT Cyclomatic Subtrees



```
UNTESTED SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM ll_compile
ROOT MODULE OF PROGRAM: ll_compile
NO DESIGN REDUCTION

UNTESTED SUBTREE #1:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance > look_ahead > get_char
 < look_ahead < advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR UNTESTED SUBTREE #1:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 204(4): current_char = '-' ==> TRUE
 get_char 58(1): end_of_file(standard_input) ==> TRUE
 advance 206(6): look_char /= '-' ==> TRUE
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


UNTESTED SUBTREE #2:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [get] < readgram > [skip_line]
 < readgram > [close] < readgram < llmain > parse > llfind < parse > llfind
 < parse > llnexttoken > advance > look_ahead > get_char < look_ahead
 < advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR UNTESTED SUBTREE #2:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> TRUE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
```

Figure 32-23. BAT Untested Cyclomatic Subtrees

```
llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
llfind 148(3): low /= high ==> TRUE
llfind 150(6;: item < llsymboltable(midpoint).key ==> TRUE
llfind 148(3): low /= high ==> TRUE
llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
advance 204(4): current_char = '-' ==> TRUE
get_char 58(1): end_of_file(standard_input) ==> TRUE
advance 206(6): look_char /= '-' ==> TRUE
advance 212(14): current_char = ascii.eot ==> TRUE
llnexttoken 342(2): lleotoks ==> FALSE
parse 680(13): lltop /= 0 ==> FALSE
parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


UNTESTED SUBTREE #3:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [get] < readgram > [get]
 < readgram > [skip_line] < readgram > buildright < readgram > buildselect
 > [skip_line] < buildselect < readgram > [close] < readgram < llmain > parse
 > llfind < parse > llfind < parse > llnexttoken > advance > look_ahead
 > get_char < look_ahead < advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR UNTESTED SUBTREE #3:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> TRUE
 buildright 397(3): i in thisrhs + 1 .. thisrhs + productions(whichprod).cardrhs ==> F
 buildselect 455(2): i in 1 .. productions(whichprod).cardsel ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 204(4): current_char = '-' ==> TRUE
 get_char 58(1): end_of_file(standard_input) ==> TRUE
 advance 206(6): look_char /= '-' ==> TRUE
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE
                                    ...
```

Figure 32-23. continued. BAT Untested Cyclomatic Subtrees

```
SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM ll_compile
ROOT MODULE OF PROGRAM: ll_compile


SUBTREE #1:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance < llnexttoken < parse
 < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #1:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #2:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [skip_line] < readgram
 > [close] < readgram < llmain > parse > llfind < parse > llfind < parse
 > llnexttoken > advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #2:
 readgram 466(2): i in 1 .. lltablesize ==> TRUE
 readgram 467(4): j in 1 .. llstringlength ==> FALSE
 readgram 472(10): ch = 'g' ==> TRUE
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #3:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [get] < readgram > [get] < readgram > [get]
 < readgram > [skip_line] < readgram > buildright < readgram > buildselect
 > [skip_line] < buildselect < readgram > [close] < readgram < llmain > parse
 > llfind < parse > llfind < parse > llnexttoken > advance < llnexttoken
 < parse < llmain < ll_compile
```

Figure 32-24. BAT Design Subtrees

```
END-TO-END TEST CONDITION LIST FOR SUBTREE #3:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> TRUE
 buildright 397(3): i in thisrhs + 1 .. thisrhs + productions(whichprod).cardrhs ==> F
 buildselect 455(2): i in 1 .. productions(whichprod).cardsel ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE


SUBTREE #4:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
                                 ...


SUBTREE #197:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance < llnexttoken < parse
 > testsynch > synchronize > [put] < synchronize > llprttoken > llprtstring
 > [put] < llprtstring > [put] < llprtstring < llprttoken < synchronize
 > [put] < synchronize > [put] < synchronize > [put_line] < synchronize
 < testsynch < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR SUBTREE #197:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 llfind 148(3): low /= high ==> FALSE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> TRUE
 parse 682(16): llstack(llsentptr).data.kind ==> group | literal
 parse 684(17): llstack(llsentptr).data.tableindex = llcurtok.tableindex ==> FALSE
 parse 688(20): llstack(llsentptr).data.tableindex = locofnull ==> FALSE
 parse 690(22): not llstack(llsentptr).lastchild ==> TRUE
 parse 691(23): llstack(llsentptr + 1).data.kind = patch ==> FALSE
 testsynch 654(1): llstack(llsentptr).data.synchindex = 0 ==> FALSE
 llprttoken 181(1): llcurtok.printvalue(1) in '!' .. '~' ==> TRUE
 llprtstring 170(2): i in str'range ==> FALSE
 synchronize 610(8): synchdata(i).sent /= 0 ==> FALSE
 synchronize 643(36): llcurtok.tableindex = llloceos ==> TRUE
 parse 706(37): lladvance ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE
```

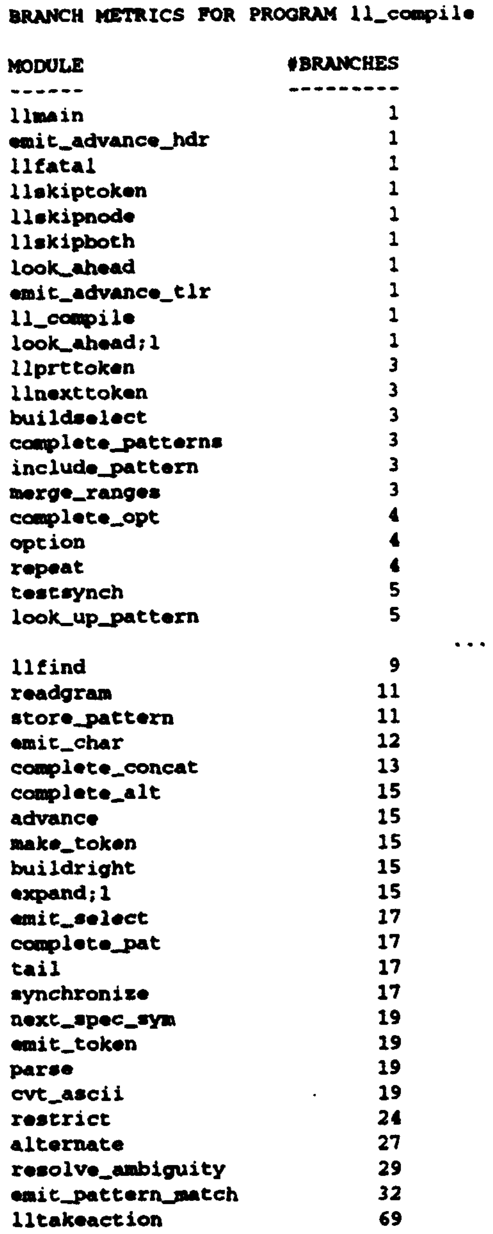
Figure 32-24. continued. BAT Design Subtrees

```
UNTESTED SUBTREES AND ASSOCIATED
INTEGRATION TEST CONDITIONS FOR PROGRAM ll_compile
ROOT MODULE OF PROGRAM: ll_compile


UNTESTED SUBTREE #1:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
 > [skip_line] < readgram > [close] < readgram < llmain > parse > llfind
 < parse > llfind < parse > llnexttoken > advance > look_ahead > get_char
 < look_ahead < advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR UNTESTED SUBTREE #1:
 readgram 466(2): i in 1 .. lltablesize ==> FALSE
 readgram 482(18): i in 1 .. llprodsize ==> FALSE
 readgram 491(27): i in 1 .. llsynchsize ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
 llfind 148(3): low /= high ==> TRUE
 llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
 llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
 llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
 advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
 advance 204(4): current_char = '-' ==> TRUE
 get_char 58(1): end_of_file(standard_input) ==> TRUE
 advance 206(6): look_char /= '-' ==> TRUE
 advance 212(14): current_char = ascii.eot ==> TRUE
 llnexttoken 342(2): lleotoks ==> FALSE
 parse 680(13): lltop /= 0 ==> FALSE
 parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE
```

Figure 32-25. BAT Untested Design Subtrees

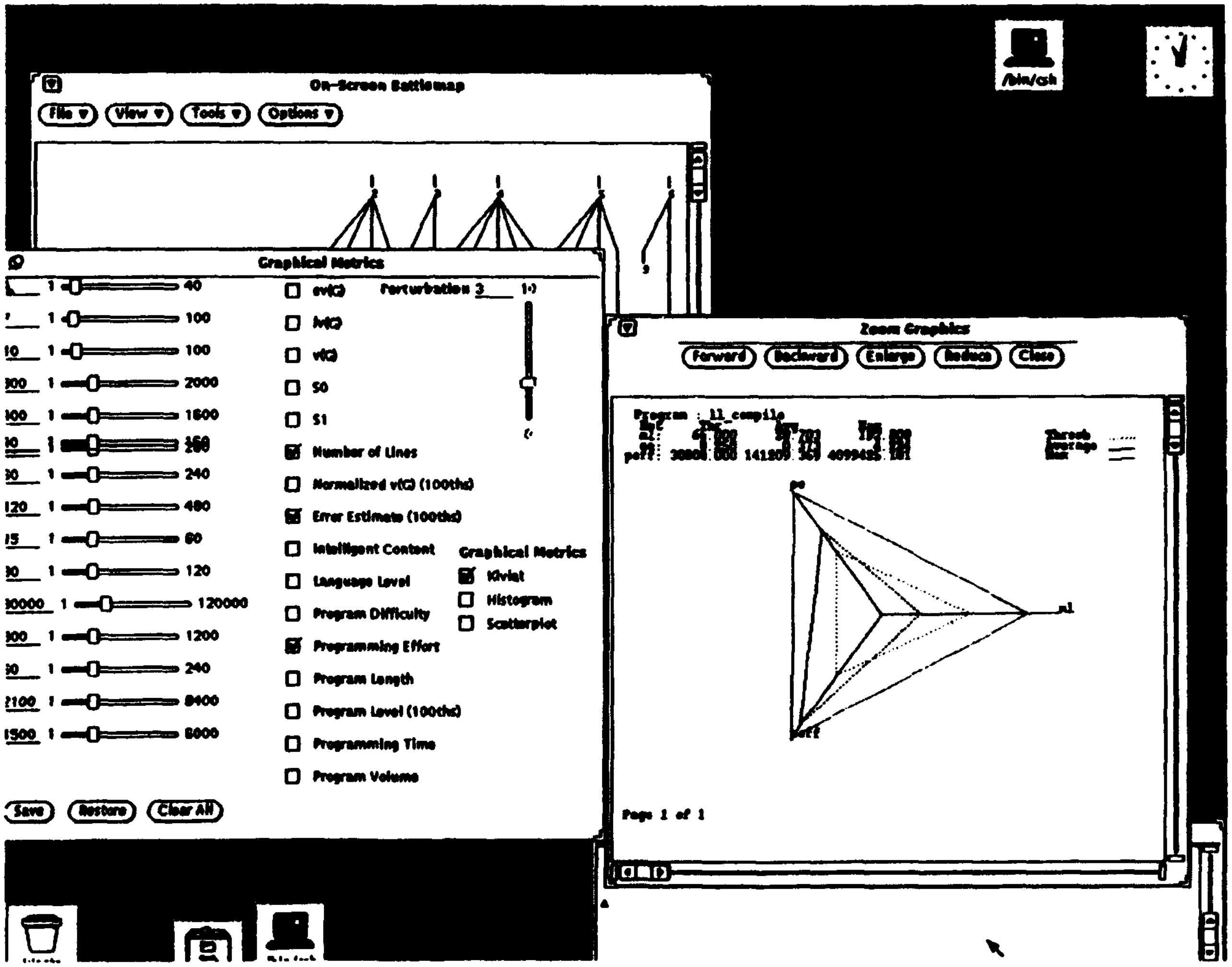
```
UNTESTED SUBTREE #2:
   ll_compile > llmain > readgram > [open] < readgram > [get] < readgram
  > [skip_line] < readgram > [get] < readgram > [get] < readgram > [skip_line]
  < readgram > [close] < readgram < llmain > parse > llfind < parse > llfind
  < parse > llnexttoken > advance > look_ahead > get_char < look_ahead
  < advance < llnexttoken < parse < llmain < ll_compile

END-TO-END TEST CONDITION LIST FOR UNTESTED SUBTREE #2:
  readgram 466(2): i in 1 .. lltablesize ==> FALSE
  readgram 482(18): i in 1 .. llprodsize ==> FALSE
  readgram 491(27): i in 1 .. llsynchsize ==> TRUE
  readgram 491(27): i in 1 .. llsynchsize ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
  llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> TRUE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> FALSE
  llfind 148(3): low /= high ==> TRUE
  llfind 150(6): item < llsymboltable(midpoint).key ==> FALSE
  llfind 152(8): item = llsymboltable(midpoint).key ==> TRUE
  llfind 153(9): llsymboltable(midpoint).kind = which ==> TRUE
  advance 200(2): (current_char = ascii.etx) or (current_char = ascii.ht) or (current_c
  advance 204(4): current_char = '-' ==> TRUE
  get_char 58(1): end_of_file(standard_input) ==> TRUE
  advance 206(6): look_char /= '-' ==> TRUE
  advance 212(14): current_char = ascii.eot ==> TRUE
  llnexttoken 342(2): lleotoks ==> FALSE
  parse 680(13): lltop /= 0 ==> FALSE
  parse 713(42): llcurtok.tableindex /= llloceos ==> FALSE
                                    ...
```

Figure 32-25. continued. BAT Untested Design Subtrees



```
BRANCH METRICS FOR PROGRAM ll_compile

MODULE                  #BRANCHES
------                  ---------
llmain                          1
emit_advance_hdr                1
llfatal                         1
llskiptoken                     1
llskipnode                      1
llskipboth                      1
look_ahead                      1
emit_advance_tlr                1
ll_compile                      1
look_ahead;1                    1
llprttoken                      3
llnexttoken                     3
buildselect                     3
complete_patterns               3
include_pattern                 3
merge_ranges                    3
complete_opt                    4
option                          4
repeat                          4
testsynch                       5
look_up_pattern                 5
                                     ...
llfind                          9
readgram                       11
store_pattern                  11
emit_char                      12
complete_concat                13
complete_alt                   15
advance                        15
make_token                     15
buildright                     15
expand;1                       15
emit_select                    17
complete_pat                   17
tail                           17
synchronize                    17
next_spec_sym                  19
emit_token                     19
parse                          19
cvt_ascii                      19
restrict                       24
alternate                      27
resolve_ambiguity              29
emit_pattern_match             32
lltakeaction                   69
```

Figure 32-26. BAT Branch Count Metrics



Graphical Metrics

```
   1 —— 40        □ ev(G)         Perturbation 3___ 10
   1 —— 100       □ iv(G)
10 1 —— 100       □ v(G)
300 1 —— 2000     □ S0
100 1 —— 1600     □ S1
10 1 —— 160       ☒ Number of Lines
30 1 —— 240       □ Normalized v(G) (100ths)
120 1 —— 480      ☒ Error Estimate (100ths)
15 1 —— 60        □ Intelligent Content      Graphical Metrics
30 1 —— 120       □ Language Level           ☒ Kiviat
30000 1 —— 120000 □ Program Difficulty       □ Histogram
300 1 —— 1200     ☒ Programming Effort       □ Scatterplot
30 1 —— 240       □ Program Length
2100 1 —— 8400    □ Program Level (100ths)
1500 1 —— 6000    □ Programming Time
                  □ Program Volume
```

```
Program : ll_compile
peff : 30800.000 141209.369 4099425.181
```

Page 1 of 1

Figure 32-27. BAT Kiviat Graph of Metrics

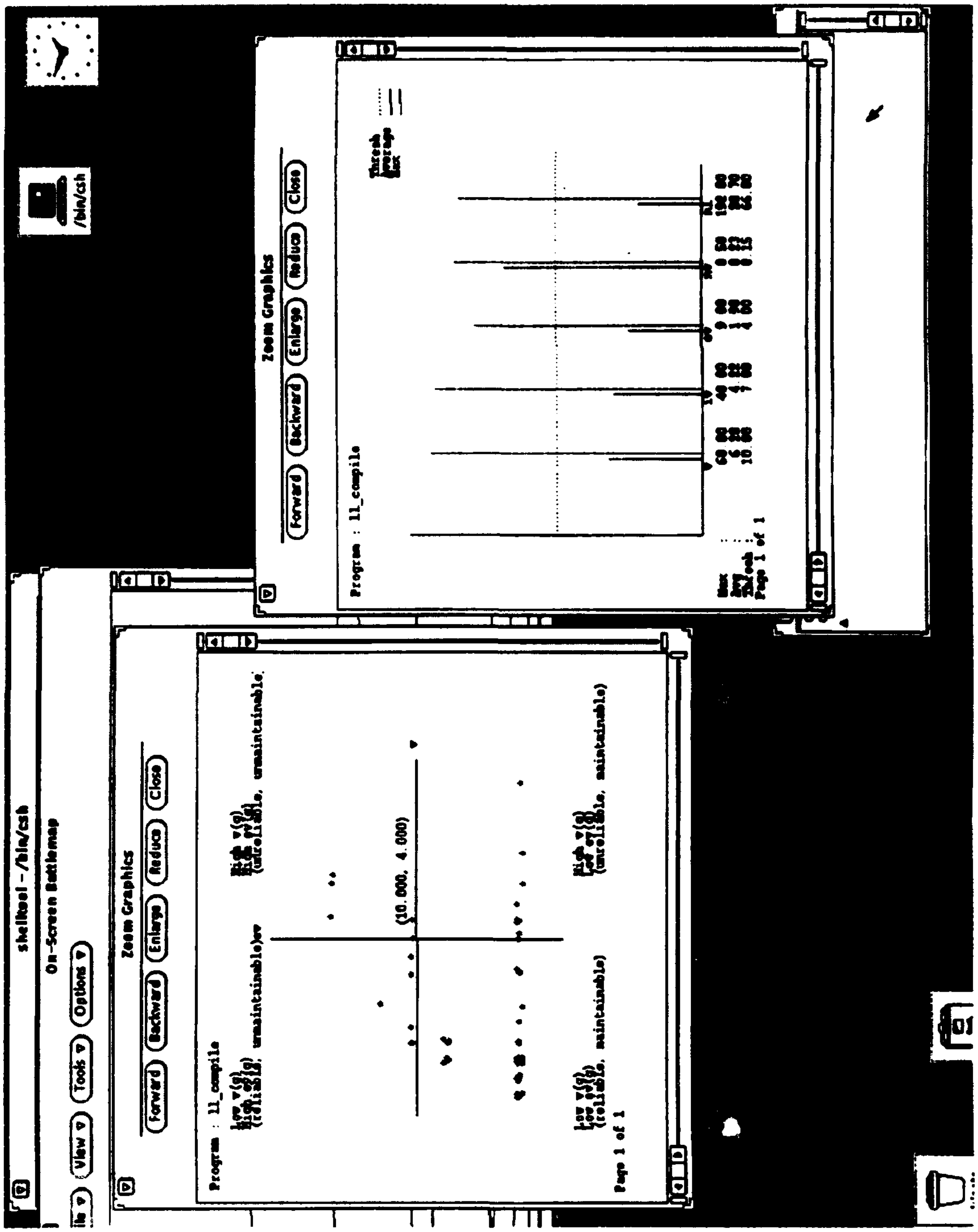
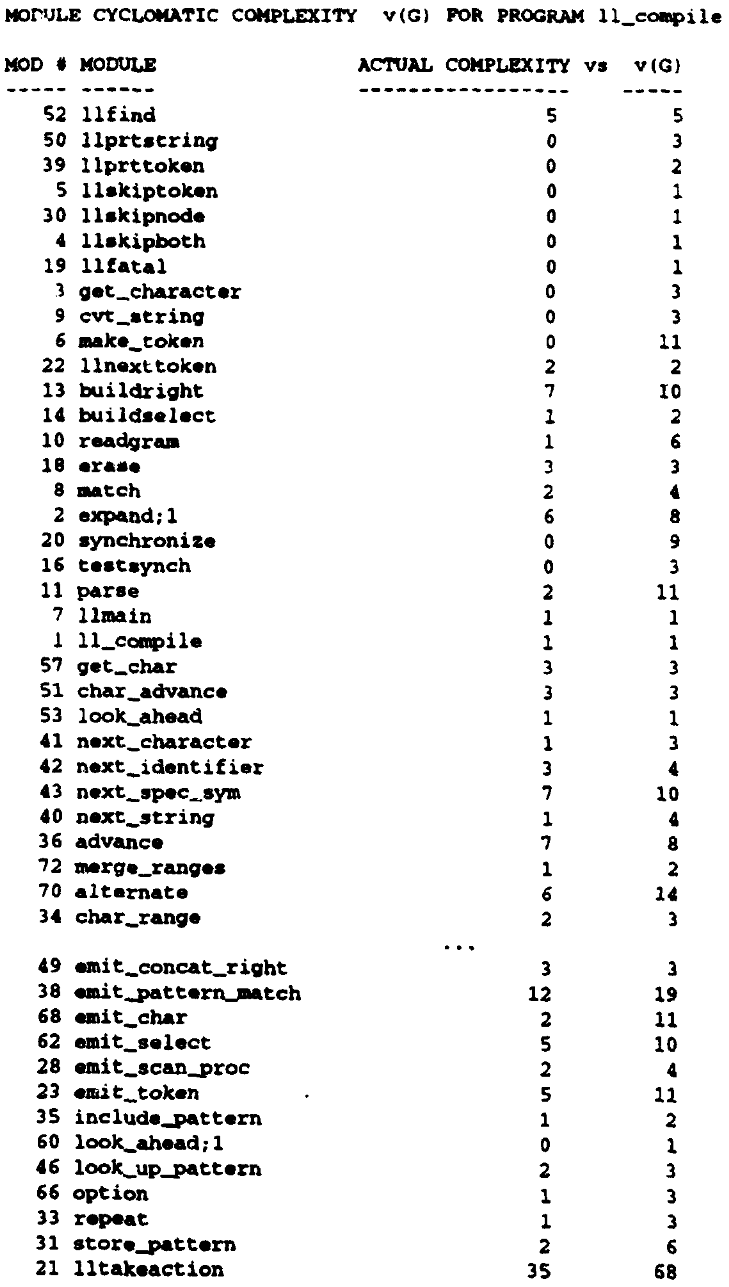
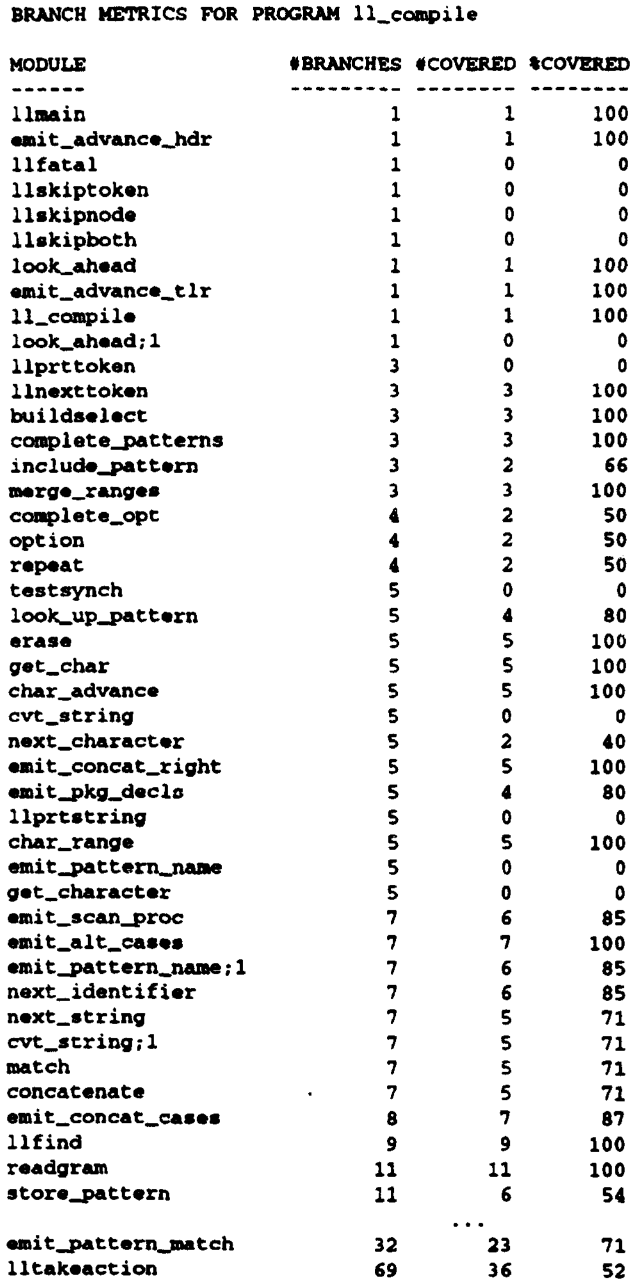


Figure 32-28. BAT Histogram and Scatterplot Metric Graphs



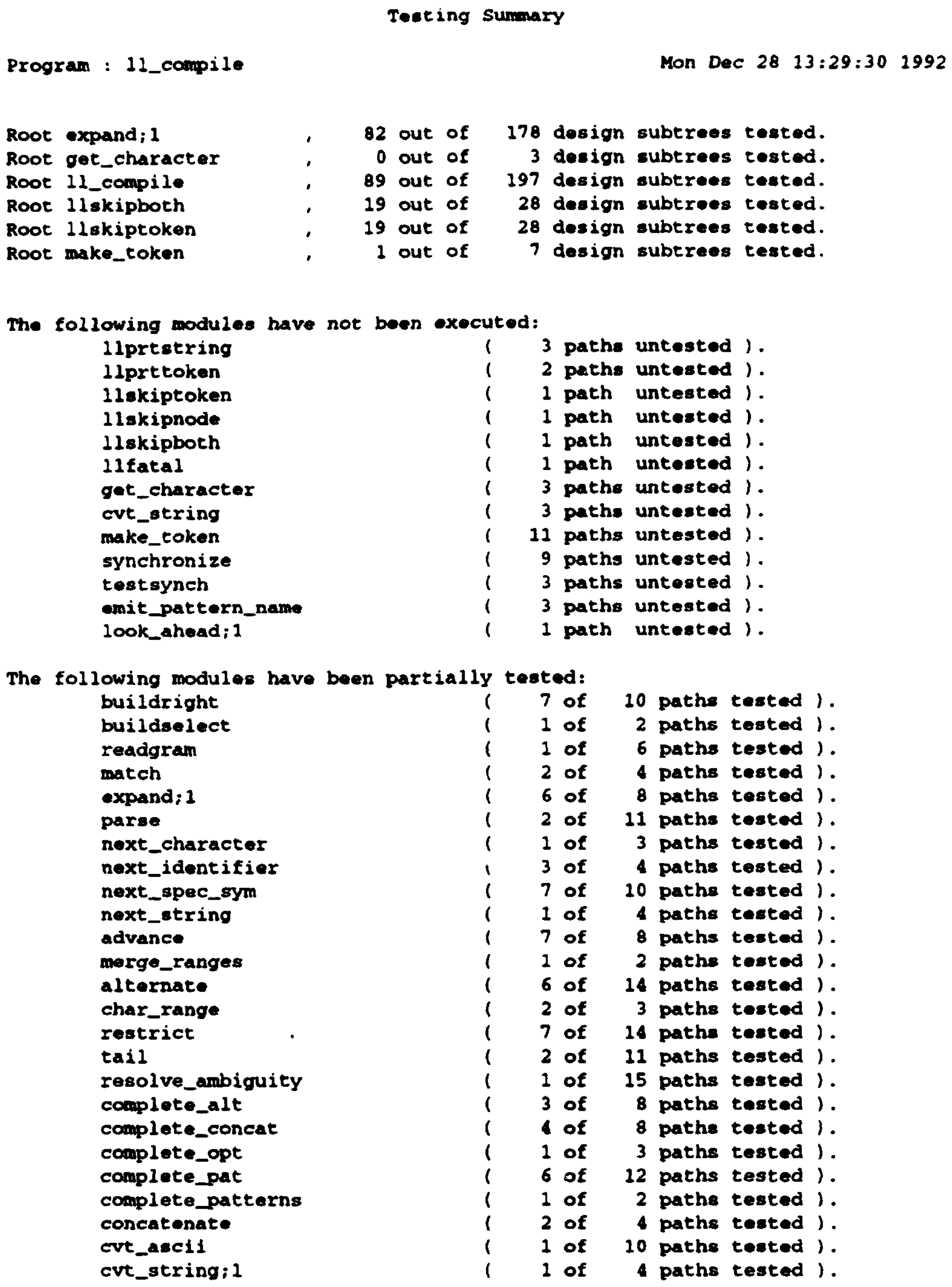
```
MODULE CYCLOMATIC COMPLEXITY  v(G) FOR PROGRAM ll_compile

MOD # MODULE                 ACTUAL COMPLEXITY vs  v(G)
----- ------                 -----------------     -----
   52 llfind                                 5         5
   50 llprtstring                            0         3
   39 llprttoken                             0         2
    5 llskiptoken                            0         1
   30 llskipnode                             0         1
    4 llskipboth                             0         1
   19 llfatal                                0         1
    3 get_character                          0         3
    9 cvt_string                             0         3
    6 make_token                             0        11
   22 llnexttoken                            2         2
   13 buildright                             7        10
   14 buildselect                            1         2
   10 readgram                               1         6
   18 erase                                  3         3
    8 match                                  2         4
    2 expand;1                               6         8
   20 synchronize                            0         9
   16 testsynch                              0         3
   11 parse                                  2        11
    7 llmain                                 1         1
    1 ll_compile                             1         1
   57 get_char                               3         3
   51 char_advance                           3         3
   53 look_ahead                             1         1
   41 next_character                         1         3
   42 next_identifier                        3         4
   43 next_spec_sym                          7        10
   40 next_string                            1         4
   36 advance                                7         8
   72 merge_ranges                           1         2
   70 alternate                              6        14
   34 char_range                             2         3
                                 ...
   49 emit_concat_right                      3         3
   38 emit_pattern_match                    12        19
   68 emit_char                              2        11
   62 emit_select                            5        10
   28 emit_scan_proc                         2         4
   23 emit_token                             5        11
   35 include_pattern                        1         2
   60 look_ahead;1                           0         1
   46 look_up_pattern                        2         3
   66 option                                 1         3
   33 repeat                                 1         3
   31 store_pattern                          2         6
   21 lltakeaction                          35        68
```

Figure 32-29. Instrumentation Tool Module Complexity for test1.lex and sample.lex



```
BRANCH METRICS FOR PROGRAM ll_compile

MODULE                    #BRANCHES #COVERED %COVERED
------                    --------- -------- --------
llmain                            1        1      100
emit_advance_hdr                  1        1      100
llfatal                           1        0        0
llskiptoken                       1        0        0
llskipnode                        1        0        0
llskipboth                        1        0        0
look_ahead                        1        1      100
emit_advance_tlr                  1        1      100
ll_compile                        1        1      100
look_ahead;1                      1        0        0
llprttoken                        3        0        0
llnexttoken                       3        3      100
buildselect                       3        3      100
complete_patterns                 3        3      100
include_pattern                   3        2       66
merge_ranges                      3        3      100
complete_opt                      4        2       50
option                            4        2       50
repeat                            4        2       50
testsynch                         5        0        0
look_up_pattern                   5        4       80
erase                             5        5      100
get_char                          5        5      100
char_advance                      5        5      100
cvt_string                        5        0        0
next_character                    5        2       40
emit_concat_right                 5        5      100
emit_pkg_decls                    5        4       80
llprtstring                       5        0        0
char_range                        5        5      100
emit_pattern_name                 5        0        0
get_character                     5        0        0
emit_scan_proc                    7        6       85
emit_alt_cases                    7        7      100
emit_pattern_name;1               7        6       85
next_identifier                   7        6       85
next_string                       7        5       71
cvt_string;1                      7        5       71
match                             7        5       71
concatenate                       7        5       71
emit_concat_cases                 8        7       87
llfind                            9        9      100
readgram                         11       11      100
store_pattern                    11        6       54
                                         ...
emit_pattern_match               32       23       71
lltakeaction                     69       36       52
```

Figure 32-30. Instrumentation Tool Branch Coverage for test1.lex and sample.lex



```
                          Testing Summary

Program : ll_compile                                Mon Dec 28 13:29:30 1992


Root expand;1          ,    82 out of   178 design subtrees tested.
Root get_character     ,     0 out of     3 design subtrees tested.
Root ll_compile        ,    89 out of   197 design subtrees tested.
Root llskipboth        ,    19 out of    28 design subtrees tested.
Root llskiptoken       ,    19 out of    28 design subtrees tested.
Root make_token        ,     1 out of     7 design subtrees tested.


The following modules have not been executed:
        llprtstring                (    3 paths untested ).
        llprttoken                 (    2 paths untested ).
        llskiptoken                (    1 path  untested ).
        llskipnode                 (    1 path  untested ).
        llskipboth                 (    1 path  untested ).
        llfatal                    (    1 path  untested ).
        get_character              (    3 paths untested ).
        cvt_string                 (    3 paths untested ).
        make_token                 (   11 paths untested ).
        synchronize                (    9 paths untested ).
        testsynch                  (    3 paths untested ).
        emit_pattern_name          (    3 paths untested ).
        look_ahead;1               (    1 path  untested ).

The following modules have been partially tested:
        buildright                 (    7 of   10 paths tested ).
        buildselect                (    1 of    2 paths tested ).
        readgram                   (    1 of    6 paths tested ).
        match                      (    2 of    4 paths tested ).
        expand;1                   (    6 of    8 paths tested ).
        parse                      (    2 of   11 paths tested ).
        next_character             (    1 of    3 paths tested ).
        next_identifier            \    3 of    4 paths tested ).
        next_spec_sym              (    7 of   10 paths tested ).
        next_string                (    1 of    4 paths tested ).
        advance                    (    7 of    8 paths tested ).
        merge_ranges               (    1 of    2 paths tested ).
        alternate                  (    6 of   14 paths tested ).
        char_range                 (    2 of    3 paths tested ).
        restrict        .          (    7 of   14 paths tested ).
        tail                       (    2 of   11 paths tested ).
        resolve_ambiguity          (    1 of   15 paths tested ).
        complete_alt               (    3 of    8 paths tested ).
        complete_concat            (    4 of    8 paths tested ).
        complete_opt               (    1 of    3 paths tested ).
        complete_pat               (    6 of   12 paths tested ).
        complete_patterns          (    1 of    2 paths tested ).
        concatenate                (    2 of    4 paths tested ).
        cvt_ascii                  (    1 of   10 paths tested ).
        cvt_string;1               (    1 of    4 paths tested ).
```

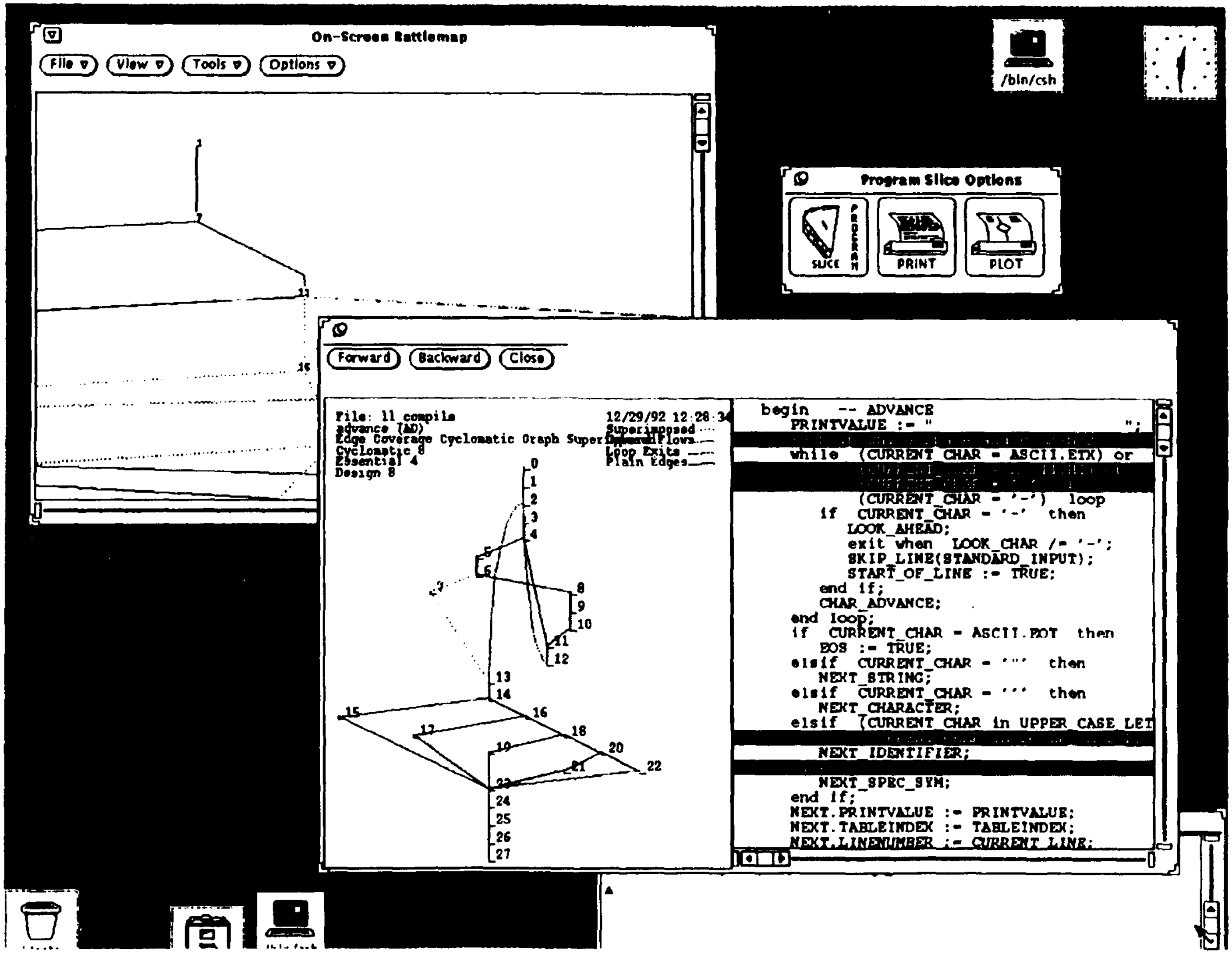
Figure 31-31. Instrumentation Tool Testing Summary for test1.lex and sample.lex

```
        emit_pkg_decls                (    1 of     3 paths tested ).
        emit_pattern_name;1           (    3 of     4 paths tested ).
        emit_alt_cases                (    3 of     4 paths tested ).
        emit_concat_cases             (    4 of     5 paths tested ).
        emit_pattern_match            (   12 of    19 paths tested ).
        emit_char                     (    2 of    11 paths tested ).
        emit_select                   (    5 of    10 paths tested ).
        emit_scan_proc                (    2 of     4 paths tested ).
        emit_token                    (    5 of    11 paths tested ).
        include_pattern               (    1 of     2 paths tested ).
        look_up_pattern               (    2 of     3 paths tested ).
        option                        (    1 of     3 paths tested ).
        repeat                        (    1 of     3 paths tested ).
        store_pattern                 (    2 of     6 paths tested ).
        lltakeaction                  (   35 of    68 paths tested ).

The following modules have full path coverage:
        llfind                        (    5 paths tested ).
        llnexttoken                   (    2 paths tested ).
        erase                         (    3 paths tested ).
        llmain                        (    1 path  tested ).
        ll_compile                    (    1 path  tested ).
        get_char                      (    3 paths tested ).
        char_advance                  (    3 paths tested ).
        look_ahead                    (    1 path  tested ).
        emit_advance_hdr              (    1 path  tested ).
        emit_advance_tlr              (    1 path  tested ).
        emit_concat_right             (    3 paths tested ).
```

Figure 32-31. continued. Instrumentation Tool Testing Summary for test1.lex and sample.lex

```
PROGRAM DESIGN COMPLEXITY FOR PROGRAM ll_compile
ROOT MODULE OF PROGRAM:                        ll_compile
DESIGN COMPLEXITY S0:                          253
TESTED DESIGN COMPLEXITY:                      137
INTEGRATION COMPLEXITY S1 (# OF SUBTREES):     197
TESTED INTEGRATION COMPLEXITY:                 89
```

Figure 32-32. Instrumentation Tool Design Integration Testedness for test1.lex and sample.lex



```
File: 11 compile                                   12/29/92 12:28:34
advance (AD)                                       Superimposed ....
Edge Coverage Cyclomatic Graph Super               Dynamic Flows ___
Cyclomatic 8                                       Loop Exits ___
Essential 4                                        Plain Edges ___
Design 8
```

```
begin    -- ADVANCE
   PRINTVALUE := "                                    ";
[illegible]
   while  (CURRENT CHAR = ASCII.ETX) or
          [illegible]
          [illegible]
          (CURRENT_CHAR = '-')  loop
      if  CURRENT_CHAR = '-'  then
         LOOK_AHEAD;
         exit when  LOOK_CHAR /= '-';
         SKIP_LINE(STANDARD_INPUT);
         START_OF_LINE := TRUE;
      end if;
      CHAR_ADVANCE;
   end loop;
   if  CURRENT_CHAR = ASCII.EOT  then
      EOS := TRUE;
   elsif  CURRENT_CHAR = '"'  then
      NEXT_STRING;
   elsif  CURRENT_CHAR = '''  then
      NEXT_CHARACTER;
   elsif  (CURRENT CHAR in UPPER CASE LET
[illegible]
      NEXT IDENTIFIER;
[illegible]
      NEXT_SPEC_SYM;
   end if;
   NEXT.PRINTVALUE := PRINTVALUE;
   NEXT.TABLEINDEX := TABLEINDEX;
   NEXT.LINENUMBER := CURRENT LINE;
```

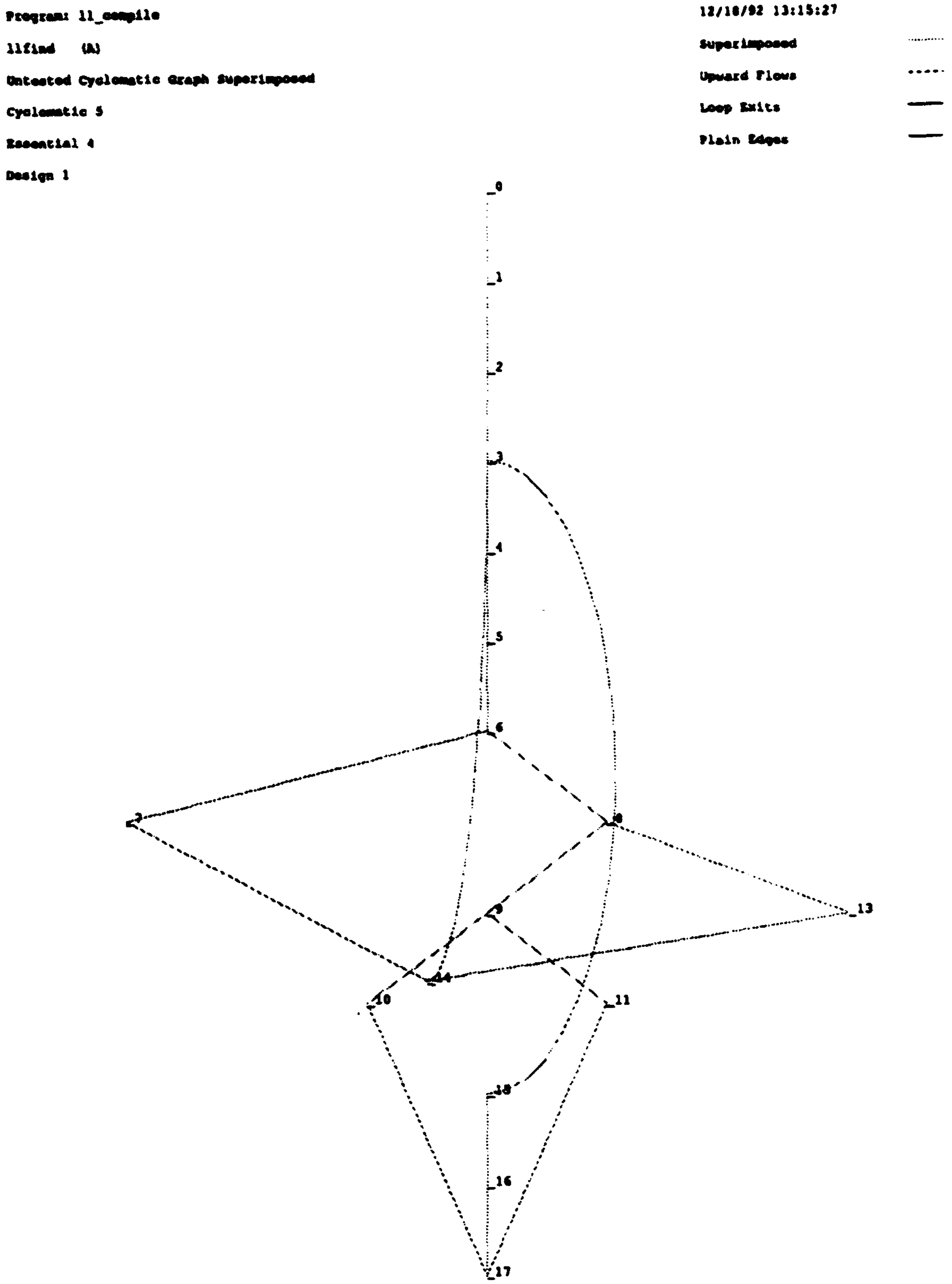
Figure 32-33. SLICE Output Using test1.lex and sample.lex

```
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      MIDPOINT := (HIGH + LOW) / 2;
      if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
         HIGH := MIDPOINT;
      elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
            return( MIDPOINT );
         else
            return( 0 );
         end if;
      else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
         LOW := MIDPOINT + 1;
      end if;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;
begin
   LL_TOKENS.ADVANCE( LLEOTOKS, LLCURTOK );
   if  LLEOTOKS  then
      LLCURTOK.PRINTVALUE := (LLSTRINGS'RANGE => ' ');
      LLCURTOK.PRINTVALUE(1..5) := "[eof]";
      LLCURTOK.TABLEINDEX := LLLOCEOS;
   end if;
end  LLNEXTTOKEN;
begin
   PRODUCTIONS(WHICHPROD).RHS := THISRHS + 1;
   CHILDCOUNT := 0;
   for  I  in  THISRHS+1 .. THISRHS+PRODUCTIONS(WHICHPROD).CARDRHS  loop
      if  I <= LLRHSSIZE  then
         THISRHS := THISRHS+1;
         GET( LLGRAM, CH );
         case  CH  is
            when  'l'  =>
               CHILDCOUNT := CHILDCOUNT+1;
               RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
               RHSARRAY(I).KIND := LITERAL;
               GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
            when  'a'  =>
               RHSARRAY(I).KIND := ACTION;
            when  'n'  =>
               CHILDCOUNT := CHILDCOUNT+1;
               RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
               RHSARRAY(I).KIND := NONTERMINAL;
               GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
            when  'g'  =>
               CHILDCOUNT := CHILDCOUNT+1;
               RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
               RHSARRAY(I).KIND := GROUP;
               GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
            when  'p'  =>
               RHSARRAY(I).KIND := PATCH;
            when others  =>
               -- the llgram table is  screwed up
               PUT( STANDARD_ERROR,
                    "**** Zuse -- Error in table file (360) ****" );
               raise  PARSING_ERROR;
         end case;
         if  END_OF_LINE( LLGRAM )  then
            RHSARRAY(I).CASEINDEX := 0;
         else
            GET( LLGRAM, RHSARRAY(I).CASEINDEX );
         end if;
         if  END_OF_LINE( LLGRAM )  then
```

Figure 32-34. SLICE Listing Excerpt

```
            RHSARRAY(I).SYNCHINDEX := 0;
         else
            GET( LLGRAM, RHSARRAY(I).SYNCHINDEX );
         end if;
         SKIP_LINE( LLGRAM );
      else
         -- llgram table is screwed up
         PUT_LINE( STANDARD_ERROR,
                   "**** Zuse -- Error in table file (372) ****" );
         -- This is a catastrophic error -- the grammar used to
         -- generate the compiler probably contained errors.
         raise  PARSING_ERROR;
      end if;
   end loop;
end  BUILDRIGHT;
begin
   PRODUCTIONS(WHICHPROD).SELSET := (others => FALSE);  -- empty set
   for  I  in  1 .. PRODUCTIONS(WHICHPROD).CARDSEL  loop
      GET( LLGRAM, TABLEINDEX );
      PRODUCTIONS(WHICHPROD).SELSET(TABLEINDEX) := TRUE;
   end loop;
   SKIP_LINE( LLGRAM );
end  BUILDSELECT;
begin   -- READGRAM
   OPEN( LLGRAM, IN_FILE, "TABLE" );
   -- read in symbol tables
   for  I  in  1 .. LLTABLESIZE  loop
      for  J  in  1 .. LLSTRINGLENGTH  loop
         GET( LLGRAM, LLSYMBOLTABLE(I).KEY(J) );
      end loop;
      GET( LLGRAM, CH );
      SKIP_LINE( LLGRAM );
      if  CH = 'g'  then
         LLSYMBOLTABLE(I).KIND := GROUP;
      else   -- assume ch = l
         LLSYMBOLTABLE(I).KIND := LITERAL;
      end if;
   end loop;
   -- read in grammar
   THISRHS := 0;
   GET( LLGRAM, AXIOM );
   SKIP_LINE( LLGRAM );
   for  I  in  1 .. LLPRODSIZE  loop
      GET( LLGRAM, PRODUCTIONS(I).LHS );
      GET( LLGRAM, PRODUCTIONS(I).CARDRHS );
      GET( LLGRAM, PRODUCTIONS(I).CARDSEL );
      SKIP_LINE( LLGRAM );
      BUILDRIGHT(I);
      BUILDSELECT(I);
   end loop;
   -- now read in synchronization info
   for  I  in  1 .. LLSYNCHSIZE  loop
      GET( LLGRAM, SYNCHDATA(I).TOKEN );   -- llsymboltable location
      GET( LLGRAM, SYNCHDATA(I).SENT );   -- symbol to skip to
      SKIP_LINE( LLGRAM );
   end loop;
   CLOSE( LLGRAM );
end  READGRAM;
begin
   -- only erase if at farthest point to the right in a production
   while  LLSTACK(LLSENTPTR).LASTCHILD  loop
      -- erase rhs
      LLSENTPTR := LLSTACK(LLSENTPTR).PARENT;
      if  LLSENTPTR = 0  then    -- stack is empty
         LLTOP := 0;
         LLADVANCE := FALSE;   -- don't try to advance beyond axiom
```

Figure 32-34. continued. SLICE Listing Excerpt



Program: ll_compile

llfind (A)

Untested Cyclomatic Graph Superimposed

Cyclomatic 5

Essential 4

Design 1

12/10/92 13:15:27

Figure 32-35. SLICE Plotted Excerpt

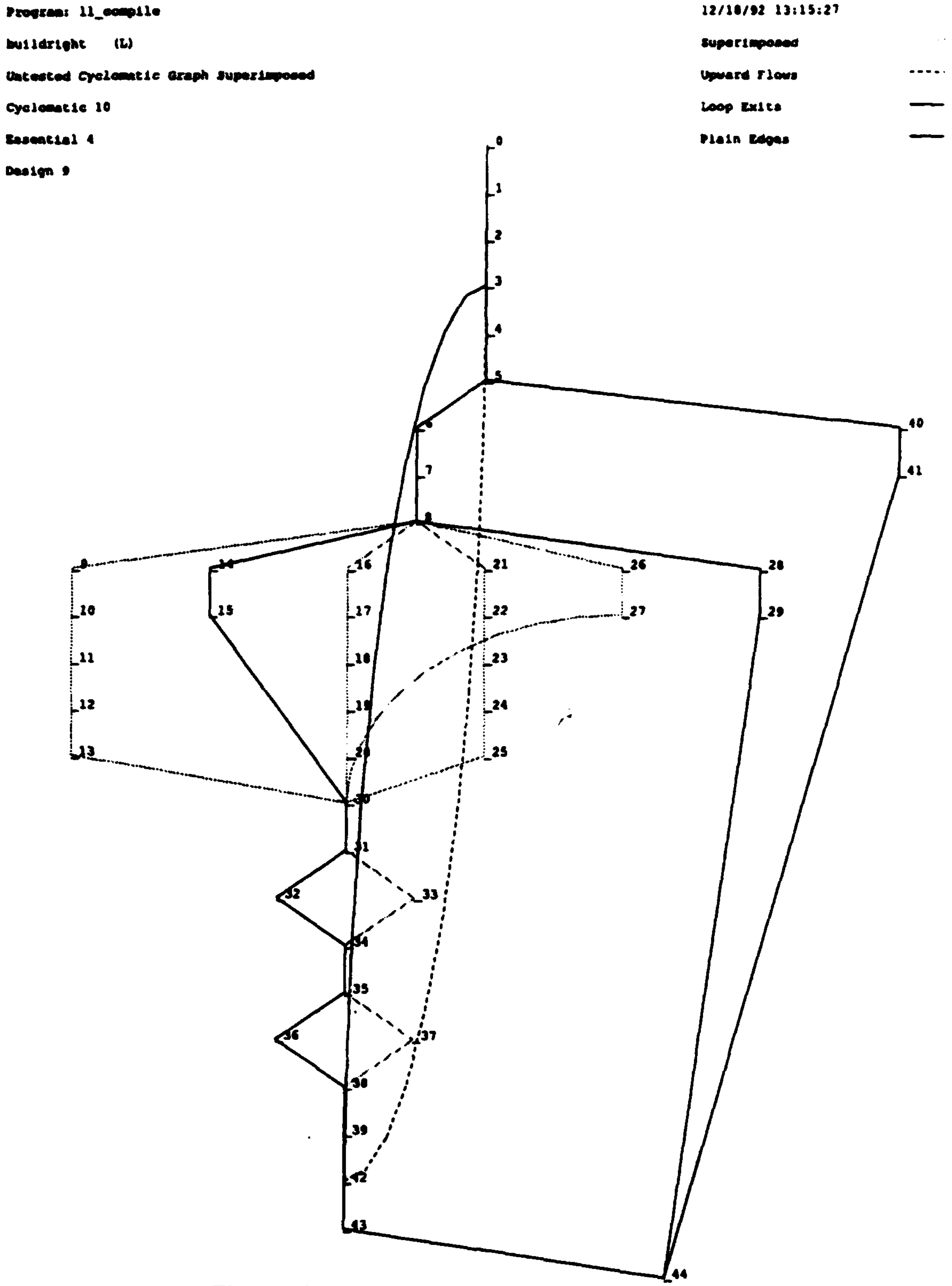
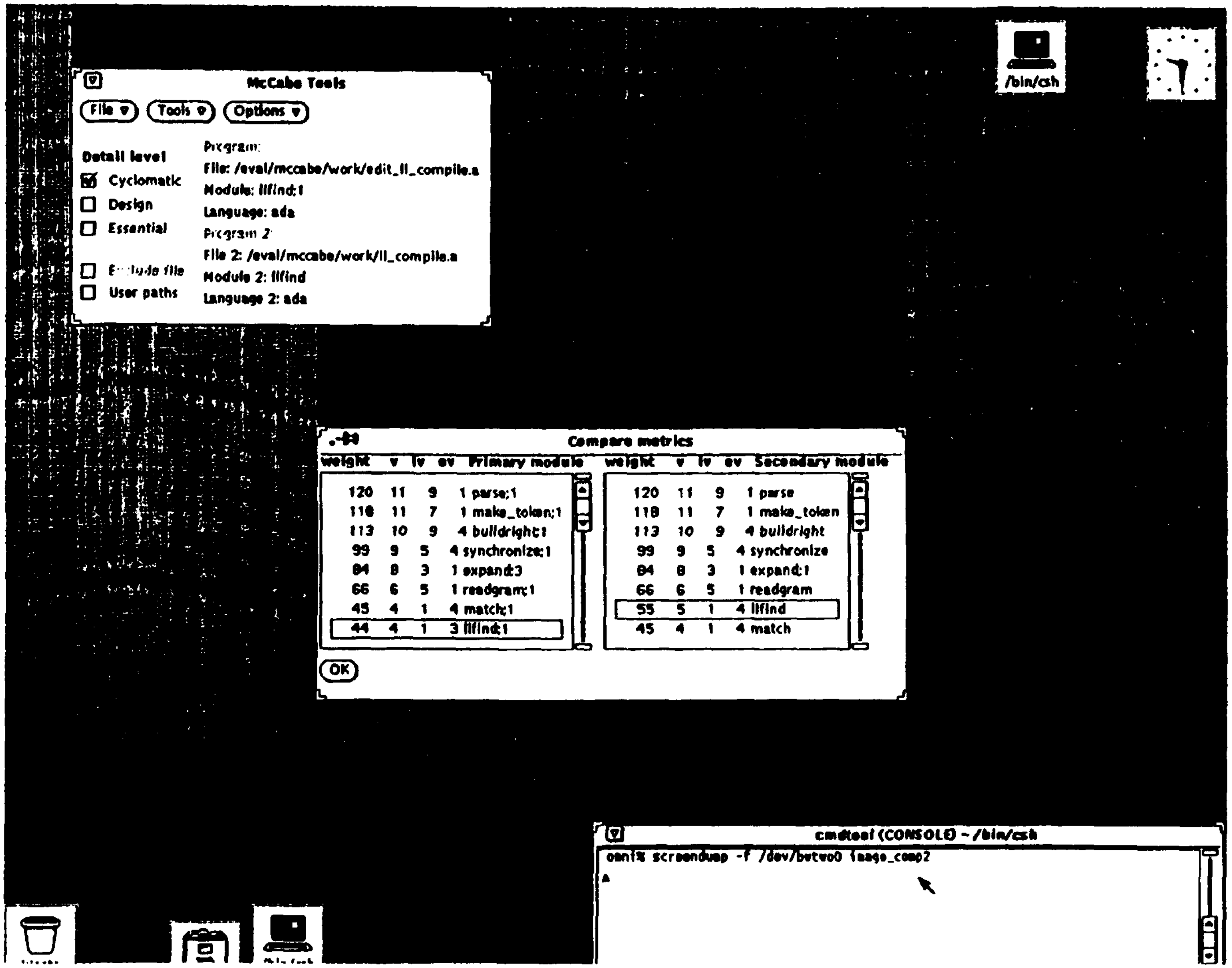


Figure 32-35. continued. SLICE Plotted Excerpt

```
Original LLFIND in file ll_compile.a:

function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
   --  Find item in symbol table -- return index or 0 if not found.
   --  Assumes symbol table is sorted in ascending order.
   LOW, MIDPOINT, HIGH: INTEGER;
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      MIDPOINT := (HIGH + LOW) / 2;
      if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
         HIGH := MIDPOINT;
      elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
            return( MIDPOINT );
         else
            return( 0 );
         end if;
      else  --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
         LOW := MIDPOINT + 1;
      end if;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;
```

```
Modified LLFIND in file edit_ll_compile.a:
From edit_l_compile.a:

function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
   --  Find item in symbol table -- return index or 0 if not found.
   --  Assumes symbol table is sorted in ascending order.
   LOW, MIDPOINT, HIGH: INTEGER;
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      if  ITEM = LLSYMBOLTABLE(LOW).KEY  then
         if  LLSYMBOLTABLE(LOW).KIND = WHICH  then
            return( LOW );
         else
            return( 0 );
         end if;
      end if;
      LOW := LOW + 1;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;
```

Figure 32-36. CodeBreaker Original and Modified Module LLFIND



McCabe Tools

Detail level

☒ Cyclomatic
□ Design
□ Essential

□ Exclude file
□ User paths

Program:
File: /eval/mccabe/work/edit_ll_compile.a
Module: llfind:1
Language: ada
Program 2:
File 2: /eval/mccabe/work/ll_compile.a
Module 2: llfind
Language 2: ada

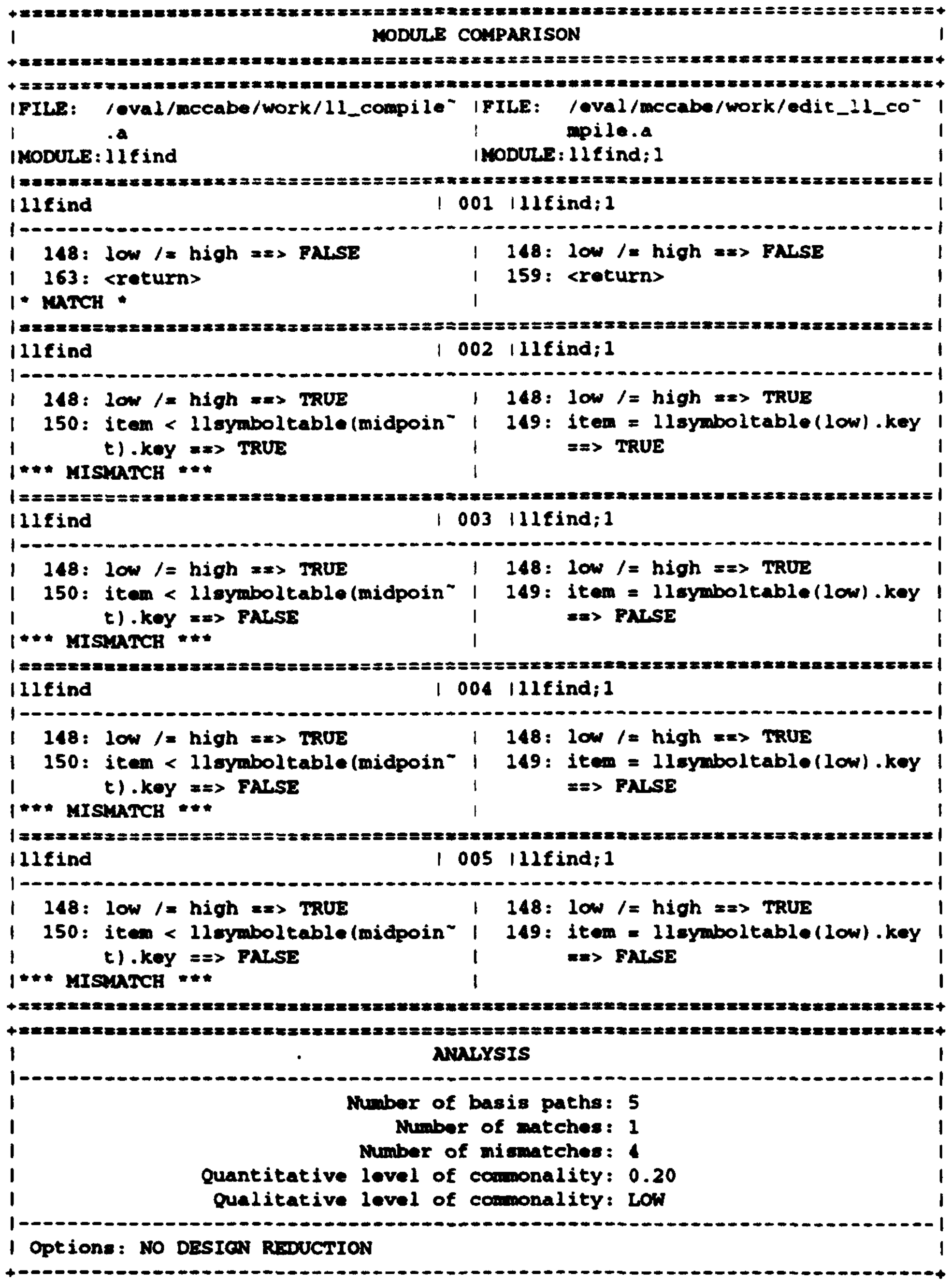
Compare metrics

```
weight   v   lv   ev  Primary module       weight   v   lv   ev  Secondary module
  120   11    9    1  parse;1                120   11    9    1  parse
  118   11    7    1  make_token;1           118   11    7    1  make_token
  113   10    9    4  buildright;1           113   10    9    4  buildright
   99    9    5    4  synchronize;1           99    9    5    4  synchronize
   84    8    3    1  expand;3                84    8    3    1  expand;1
   66    6    5    1  readgram;1              66    6    5    1  readgram
   45    4    1    4  match;1                 55    5    1    4  llfind
   44    4    1    3  llfind;1                45    4    1    4  match
```

cmdtool (CONSOLE) ~/bin/csh

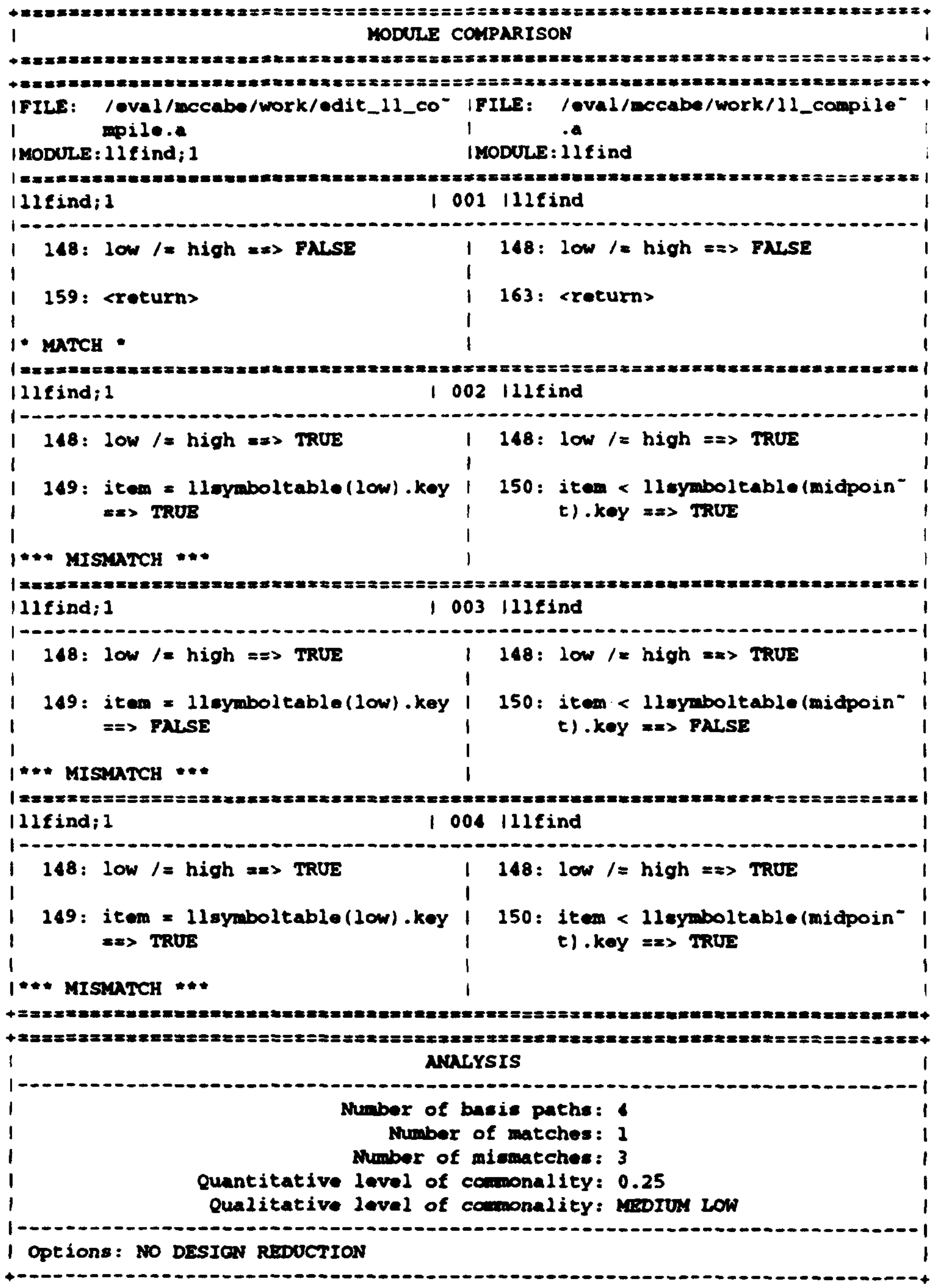
```
omni% screendump -f /dev/bwtwo0 image_comp2
```

Figure 32-37. CodeBreaker Comparison Metrics for LLFIND



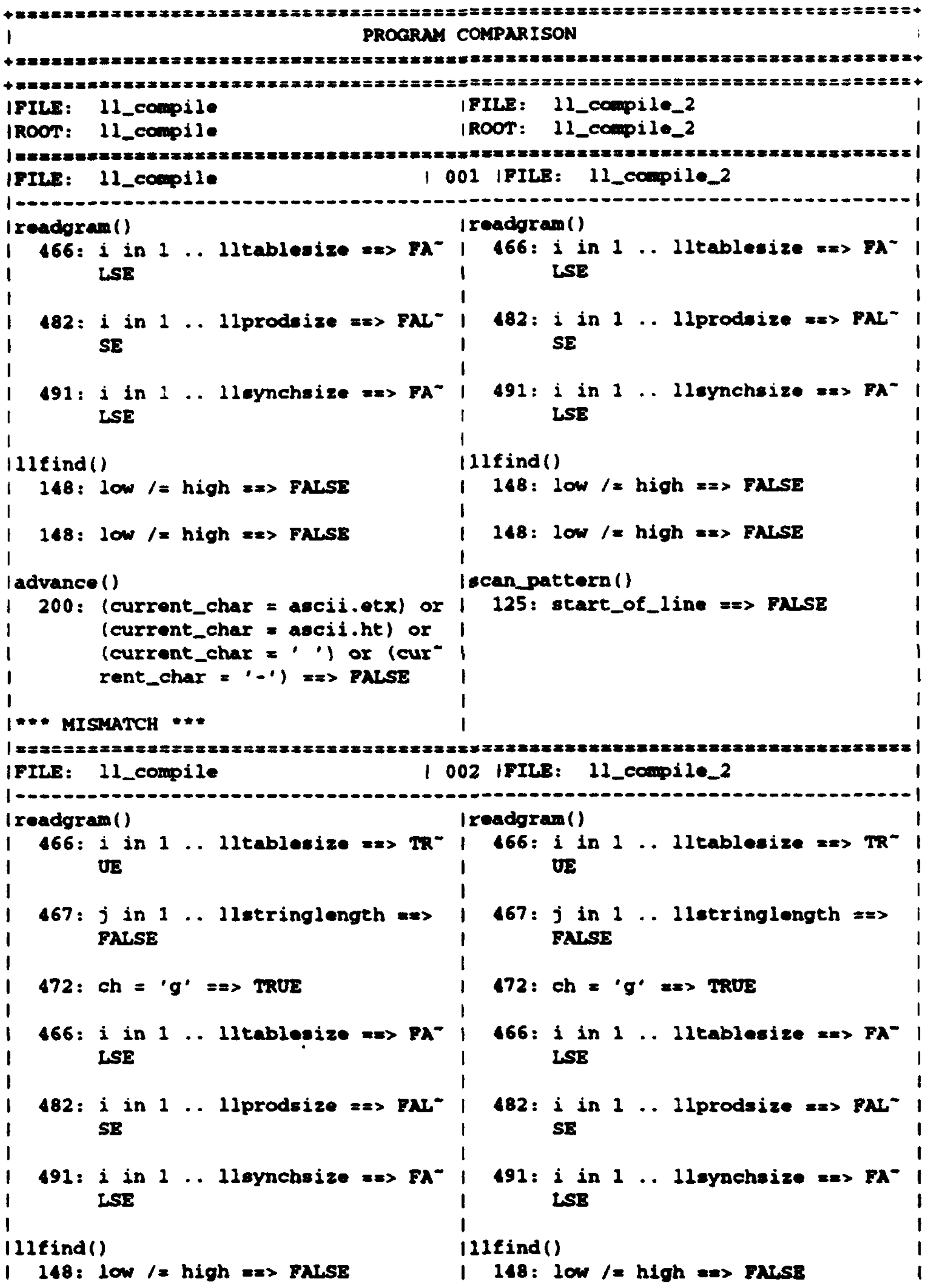
```
+==========================================================================+
|                             MODULE COMPARISON                            |
+==========================================================================+
+==========================================================================+
|FILE:  /eval/mccabe/work/ll_compile~ |FILE:  /eval/mccabe/work/edit_ll_co~ |
|       .a                            |       mpile.a                       |
|MODULE:llfind                        |MODULE:llfind;1                      |
|==========================================================================|
|llfind                          | 001 |llfind;1                            |
|--------------------------------------------------------------------------|
|  148: low /= high ==> FALSE         |  148: low /= high ==> FALSE         |
|  163: <return>                      |  159: <return>                      |
|* MATCH *                            |                                     |
|==========================================================================|
|llfind                          | 002 |llfind;1                            |
|--------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|  150: item < llsymboltable(midpoin~ |  149: item = llsymboltable(low).key |
|       t).key ==> TRUE               |       ==> TRUE                      |
|*** MISMATCH ***                     |                                     |
|==========================================================================|
|llfind                          | 003 |llfind;1                            |
|--------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|  150: item < llsymboltable(midpoin~ |  149: item = llsymboltable(low).key |
|       t).key ==> FALSE              |       ==> FALSE                     |
|*** MISMATCH ***                     |                                     |
|==========================================================================|
|llfind                          | 004 |llfind;1                            |
|--------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|  150: item < llsymboltable(midpoin~ |  149: item = llsymboltable(low).key |
|       t).key ==> FALSE              |       ==> FALSE                     |
|*** MISMATCH ***                     |                                     |
|==========================================================================|
|llfind                          | 005 |llfind;1                            |
|--------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|  150: item < llsymboltable(midpoin~ |  149: item = llsymboltable(low).key |
|       t).key ==> FALSE              |       ==> FALSE                     |
|*** MISMATCH ***                     |                                     |
+==========================================================================+
+==========================================================================+
|                                 ANALYSIS                                 |
|--------------------------------------------------------------------------|
|                          Number of basis paths: 5                        |
|                              Number of matches: 1                        |
|                           Number of mismatches: 4                        |
|               Quantitative level of commonality: 0.20                    |
|                Qualitative level of commonality: LOW                     |
|--------------------------------------------------------------------------|
| Options: NO DESIGN REDUCTION                                             |
+--------------------------------------------------------------------------+
```

Figure 32-38. CodeBreaker Comparison of Modified Against Original LLFIND



```
+===========================================================================+
|                           MODULE COMPARISON                               |
+===========================================================================+
+===========================================================================+
|FILE:  /eval/mccabe/work/edit_ll_co~ |FILE:  /eval/mccabe/work/ll_compile~ |
|       mpile.a                       |       .a                            |
|MODULE:llfind;1                      |MODULE:llfind                        |
|===========================================================================|
|llfind;1                       | 001 |llfind                               |
|---------------------------------------------------------------------------|
|  148: low /= high ==> FALSE         |  148: low /= high ==> FALSE         |
|                                     |                                     |
|  159: <return>                      |  163: <return>                      |
|                                     |                                     |
|* MATCH *                            |                                     |
|===========================================================================|
|llfind;1                       | 002 |llfind                               |
|---------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|                                     |                                     |
|  149: item = llsymboltable(low).key |  150: item < llsymboltable(midpoin~ |
|       ==> TRUE                      |       t).key ==> TRUE               |
|                                     |                                     |
|*** MISMATCH ***                     |                                     |
|===========================================================================|
|llfind;1                       | 003 |llfind                               |
|---------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|                                     |                                     |
|  149: item = llsymboltable(low).key |  150: item < llsymboltable(midpoin~ |
|       ==> FALSE                     |       t).key ==> FALSE              |
|                                     |                                     |
|*** MISMATCH ***                     |                                     |
|===========================================================================|
|llfind;1                       | 004 |llfind                               |
|---------------------------------------------------------------------------|
|  148: low /= high ==> TRUE          |  148: low /= high ==> TRUE          |
|                                     |                                     |
|  149: item = llsymboltable(low).key |  150: item < llsymboltable(midpoin~ |
|       ==> TRUE                      |       t).key ==> TRUE               |
|                                     |                                     |
|*** MISMATCH ***                     |                                     |
+===========================================================================+
+===========================================================================+
|                                ANALYSIS                                   |
|---------------------------------------------------------------------------|
|                           Number of basis paths: 4                        |
|                               Number of matches: 1                        |
|                            Number of mismatches: 3                        |
|                Quantitative level of commonality: 0.25                    |
|                 Qualitative level of commonality: MEDIUM LOW              |
|---------------------------------------------------------------------------|
| Options: NO DESIGN REDUCTION                                              |
+---------------------------------------------------------------------------+
```

Figure 32-39. CodeBreaker Comparison of Original Against Modified LLFIND



```
+==========================================================================+
|                           PROGRAM COMPARISON                             |
+==========================================================================+
+==========================================================================+
|FILE:  ll_compile                     |FILE:  ll_compile_2                |
|ROOT:  ll_compile                     |ROOT:  ll_compile_2                |
|==========================================================================|
|FILE:  ll_compile                | 001 |FILE:  ll_compile_2               |
|--------------------------------------------------------------------------|
|readgram()                            |readgram()                         |
|  466: i in 1 .. lltablesize ==> FA~  |  466: i in 1 .. lltablesize ==> FA~ |
|       LSE                            |       LSE                         |
|                                      |                                   |
|  482: i in 1 .. llprodsize ==> FAL~  |  482: i in 1 .. llprodsize ==> FAL~ |
|       SE                             |       SE                          |
|                                      |                                   |
|  491: i in 1 .. llsynchsize ==> FA~  |  491: i in 1 .. llsynchsize ==> FA~ |
|       LSE                            |       LSE                         |
|                                      |                                   |
|llfind()                              |llfind()                           |
|  148: low /= high ==> FALSE          |  148: low /= high ==> FALSE       |
|                                      |                                   |
|  148: low /= high ==> FALSE          |  148: low /= high ==> FALSE       |
|                                      |                                   |
|advance()                             |scan_pattern()                     |
|  200: (current_char = ascii.etx) or  |  125: start_of_line ==> FALSE     |
|       (current_char = ascii.ht) or   |                                   |
|       (current_char = ' ') or (cur~  |                                   |
|       rent_char = '-') ==> FALSE     |                                   |
|                                      |                                   |
|*** MISMATCH ***                      |                                   |
|==========================================================================|
|FILE:  ll_compile                | 002 |FILE:  ll_compile_2               |
|--------------------------------------------------------------------------|
|readgram()                            |readgram()                         |
|  466: i in 1 .. lltablesize ==> TR~  |  466: i in 1 .. lltablesize ==> TR~ |
|       UE                             |       UE                          |
|                                      |                                   |
|  467: j in 1 .. llstringlength ==>   |  467: j in 1 .. llstringlength ==> |
|       FALSE                          |       FALSE                       |
|                                      |                                   |
|  472: ch = 'g' ==> TRUE              |  472: ch = 'g' ==> TRUE           |
|                                      |                                   |
|  466: i in 1 .. lltablesize ==> FA~  |  466: i in 1 .. lltablesize ==> FA~ |
|       LSE                            |       LSE                         |
|                                      |                                   |
|  482: i in 1 .. llprodsize ==> FAL~  |  482: i in 1 .. llprodsize ==> FAL~ |
|       SE                             |       SE                          |
|                                      |                                   |
|  491: i in 1 .. llsynchsize ==> FA~  |  491: i in 1 .. llsynchsize ==> FA~ |
|       LSE                            |       LSE                         |
|                                      |                                   |
|llfind()                              |llfind()                           |
|  148: low /= high ==> FALSE          |  148: low /= high ==> FALSE       |
```

Figure 32-40. CodeBreaker Program Comparison

```
|                                        |                                     |
|  148: low /= high ==> FALSE            |  148: low /= high ==> FALSE         |
|                                        |                                     |
|advance()                               |scan_pattern()                       |
|  200: (current_char = ascii.etx) or    |  125: start_of_line ==> FALSE       |
|       (current_char = ascii.ht) or     |                                     |
|       (current_char = ' ') or (cur~    |                                     |
|       rent_char = '-') ==> FALSE       |                                     |
|                                        |                                     |
|*** MISMATCH ***                        |                                     |
|==============================================================================|
|FILE:  ll_compile                 | 003 |FILE:  ll_compile_2                  |
|------------------------------------------------------------------------------|
|readgram()                              |readgram()                           |
|  466: i in 1 .. lltablesize ==> FA~    |  466: i in 1 .. lltablesize ==> FA~ |
|       LSE                              |       LSE                           |
                                        ...
|==============================================================================|
|FILE:  ll_compile                 | 197 |FILE:  ll_compile_2                  |
|------------------------------------------------------------------------------|
|readgram()                              |readgram()                           |
|  466: i in 1 .. lltablesize ==> FA~    |  466: i in 1 .. lltablesize ==> FA~ |
|       LSE                              |       LSE                           |
|                                        |                                     |
|  482: i in 1 .. llprodsize ==> FAL~    |  482: i in 1 .. llprodsize ==> FAL~ |
|       SE                               |       SE                            |
|                                        |                                     |
|  491: i in 1 .. llsynchsize ==> FA~    |  491: i in 1 .. llsynchsize ==> FA~ |
|       LSE                              |       LSE                           |
|                                        |                                     |
|llfind()                                |llfind()                             |
|  148: low /= high ==> FALSE            |  148: low /= high ==> FALSE         |
|                                        |                                     |
|  148: low /= high ==> FALSE            |  148: low /= high ==> FALSE         |
|                                        |                                     |
|advance()                               |scan_pattern()                       |
|  200: (current_char = ascii.etx) or    |  125: start_of_line ==> FALSE       |
|       (current_char = ascii.ht) or     |                                     |
|       (current_char = ' ') or (cur~    |                                     |
|       rent_char = '-') ==> FALSE       |                                     |
|                                        |                                     |
|*** MISMATCH ***                        |                                     |
+==============================================================================+
+==============================================================================+
|                                  ANALYSIS                                    |
|------------------------------------------------------------------------------|
|                                  Number of subtrees: 197                     |
|                                   Number of matches: 0                       |
|                                Number of mismatches: 197                     |
|                     Quantitative level of commonality: 0.00                  |
|                      Qualitative level of commonality: VERY LOW              |
|------------------------------------------------------------------------------|
| Options: IGNORE MODULE NAMES  IGNORE MODULE CALLS                            |
+------------------------------------------------------------------------------+
```

Figure 32-40. continued. CodeBreaker Program Comparison

REFERENCES

[Halstead 1977] Halstead, M.H. 1977. *Elements of Software Science*. NY: Elsevier North-Holland Publishing.

[McCabe 1976] McCabe, T.J. 1976. "A Complexity Measure." *IEEE: Transactions on Software Engineering*, Vol. 2, No. 4 (Dec), pp. 308-320.

[McCabe 1982] McCabe, T.J. December 1982. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*. NBS Special Publication 500-99. Gaithersburg, MD: National Institute of Standards and Technology.

[Mohanty 1976] Mohanty, S.N. June 1976. *Automatic Program Testing*. Ph.D. Diss., Polytechnic Institute of New York.

[Nielsen 1988] Nielsen, K., and K. Shumate. 1988. *Designing Large Real-Time Systems with Ada*. NY: McGraw-Hill.

[RADC 1983] Rome Air Development Center. July 1983. *Software Quality Measurement for Distributed Systems*. RADC-TR-83-175.

[SPC 1991] Software Productivity Consortium. 1991. *Ada Quality and Style: Guidelines for Professional Programmers*. NY: Van Nostrand Reinhold.

[Youngblut 1992] Youngblut, C. December 1992. *An Examination of Selected Software Testing Tools: 1992*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2769.

ACRONYMS AND ABBREVIATIONS

ACT	Analysis of Complexity
ADADL	Ada-based Design and Documentation Language
ADAMA	Ada Measurement and Analysis Tool
ATA	AdaTEST Analysis
ATH	AdaTEST Harness
ATI	AdaTEST Instrumentor
BAT	Battlemap Analysis Tool
BMD	Ballistic Missile Defense
BMDO	Ballistic Missile Defense Organization
CASE	Computer-aided Software Engineering
CRWG	Computer Resources Working Group
DDTs	Distributed Defect Tracking System
DFD	Data Flow Diagram
DEC	Digital Equipment Corporation
ESD	Electronic Systems Division
GKS	Graphical Kernel System
GPALS	Global Protection Against Limited Strikes
IBM	International Business Machines
IDA	Institute for Defense Analyses
IEEE	Institute for Electrical and Electronics Engineers, Inc.
LCSAJ	Linear Code Sequence and Jump
LDRA	Liverpool Data Research Associates
MALPAS	Malvern Program Analysis Suite

PC	Personal Computer
PDL	Program Design Language
QA	Quality Assurance
RADC	Rome Air Development Center
StP	Software through Pictures
SDI	Strategic Defense Initiative
SPC	Software Productivity Consortium
START	Structured Testing and Requirements Tool
TBGEN	Test Bed Generator
TCMON	Test Coverage Monitor
TST	Test Support Tool
US	United States
VDM	Vienna Development Method

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
October 1993

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
An Examination of Selected Software Testing Tools: 1993 Supplement

5. FUNDING NUMBERS
MDA 903 89 C 0003
Task T-R2-597.21

6. AUTHOR(S)
Christine Youngblut, Bill Brykczynski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2925

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
BMDO/GSI
The Pentagon, Room 1E149
Washington, D.C. 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution: May 12, 1994.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

This paper reports on the examination of eight additional tools for testing Ada code. It is a supplement to IDA Paper P-2769, *An Examination of Selected Software Testing Tools: 1992*, which reported on 27 tools that provide for test management, problem reporting, and static and dynamic analysis of Ada code. The tool discussions provide software development managers with information that will help them gain an understanding of the current capabilities of tools that are commercially available, the functionality of these tools, and how they can aid the development and support of Ada software. The newly examined tools provide static and dynamic analysis capabilities. They were applied to sample Ada programs in order to assess their functionality. Each tool was then described in terms of its functionality, ease of use, and documentation and support. Problems encountered during the examination and other pertinent observations were also recorded. Available testing tools offer important opportunities for increasing software quality and reducing development and support costs. The wide variety of functionality provided by tools in the same category, however, and, in some cases, lack of tool maturity, require careful tool selection on behalf of a potential user.

14. SUBJECT TERMS
Software Testing Tools, Ada Programming Language, Test Management, Problem Reporting, Static and Dynamic Analysis, Software.

15. NUMBER OF PAGES
196

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR